LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

(14) MIT/LCS/TR-186

(12) B.S.

(11) Sep 77

(6)

A STRUCTURE MEMORY FOR DATA FLOW COMPUTERS.

(9) Master's thesis,

(10) William B. Ackerman

(12) 125p.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. (15) N00014-75-C-0661

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-186	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): A Structure Memory for Data Flow Computers	5. TYPE OF REPORT & PERIOD COVERED: S.M. Thesis, August 1977
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-186
7. AUTHOR(s): W. B. Ackerman	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT/Laboratory for Computer Science, 545 Technology Square, Cambridge, Ma 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, Department of Defense, 1400 Wilson Boulevard, Arlington, Virginia 22209	12. REPORT DATE: September 1977
	13. NUMBER OF PAGES: 126
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Department of the Navy, Information Systems Program, Arlington, Virginia 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

asynchronous digital systems
packet communication
data flow computers
data structures
memory
cache memory

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A data flow computer is one which achieves enormous concurrency of instruction execution through a machine architecture that acts directly on a data dependency graph of the program. To handle arrays and data structures effectively a data flow computer must have access to a memory system which can handle large numbers of concurrent transactions. This thesis presents a design for such a memory. A "cache" mechanism is presented for improving the performance of the system, and a mechanism is given for using sequential-access devices such as ...

20.

shift registers as the memory medium. The memory system design uses the "packet communication" concept, in which the components of the system communicate only through the transmission of fixed size "packets" of data.

MIT/LCS/TR-186

A STRUCTURE MEMORY FOR DATA FLOW COMPUTERS

by

WILLIAM B. ACKERMAN

August 1977

Massachusetts Institute of Technology

Laboratory for Computer Science
(formerly Project MAC)

Cambridge Massachusetts 02139

A STRUCTURE MEMORY FOR DATA FLOW COMPUTERS

by

William B. Ackerman

Submitted to the Department of Electrical Engineering and Computer Science on August 26, 1977 in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

A data flow computer is one which achieves enormous concurrency of instruction execution through a machine architecture that acts directly on a data dependency graph of the program. To handle arrays and data structures effectively, a data flow computer must have access to a memory system which can handle large numbers of concurrent transactions. This thesis presents a design for such a memory. A "cache" mechanism is presented for improving the performance of the system, and a mechanism is given for using sequential-access devices such as shift registers as the memory medium. The memory system design uses the "packet communication" concept, in which the components of the system communicate only through the transmission of fixed size "packets" of data.

THESIS SUPERVISOR: Jack B. Dennis

TITLE: Professor of Computer Science and Engineering

ACKNOWLEDGMENTS

I wish to thank Professor Jack Dennis for his encouragement and support through this research and for providing an intellectually stimulating environment in the Computation Structures Group. I would like to thank Glen Miranker and Lynn Montz for their helpful comments on parts of this thesis. The Laboratory for Computer Science provided facilities for the preparation of this thesis.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

TABLE OF CONTENTS

0.0 INTRODUCTION

A data flow computer is a machine with architecture radically different from that of existing computers. It can perform computations simultaneously on many different parts of a program. A typical data flow computer has many arithmetic processors, and can utilize all of them simultaneously, each executing a different instruction.

To handle arrays and other data structures, a data flow computer must have a data structure processing facility and memory that has a similar facility to perform many operations concurrently. Such a data structure memory is the subject of this thesis.

A data flow computer owes its great speed to its ability to perform many operations at once, even though each individual operation is no faster than on a conventional computer. The same is true of the memory. The memory to be presented here has a retrieval delay just as great as conventional memories, since no new circuit technology will be proposed. However, it has an enormous data transfer rate because of its ability to handle concurrent transactions. This concurrency is made possible by an unusual type of interface called packet communication.

Section 1 of this thesis is an overview of data flow computers and the type of memory that such a computer requires for structure processing. Section 2 is a treatment of packet communication systems, showing how their behavior is defined. In section 3 the basic memory unit is described, along with a "cache" mechanism and an "interleaving" method to improve its performance. In section 4 an implementation of the memory using shift registers or magnetic disks will be given, showing how the disadvantages of such devices can be overcome through the use of packet communication. Section 5 examines some aspects of the processing unit that uses the memory, and section 6 examines the "deadlock" problem and the cost of overcoming it. Section 7 presents suggestions for future research.

1.0 DATA FLOW COMPUTERS

As the need increases for ever faster computers, one technique for improving performance that has drawn considerable interest in the last few years is a radically new design known as a data flow computer [6] [7] [11] [15]. A conventional computer has only one locus of control, that is, one point in the program at any given instant at which instructions are executed. Ability to execute more than one instruction at a time can improve performance significantly, and some computers use an instruction lookahead to achieve this [3] [9]. However, the benefits of lookahead methods are limited, and such computers are enormously complex. Other attempts to increase instruction concurrency include "array processors" [16], but such machines are inflexible and extremely difficult to program.

A data flow computer achieves executional concurrency by using a different internal representation of the source program. Instead of representing the program as a list of instructions to be executed in a particular order, the program is represented as a data flow schema. A data flow schema is a directed graph whose nodes represent instructions and whose arcs show the data dependence among instructions. The order of instruction execution is determined solely by the data dependence - an instruction is executed when all of its data sources have produced results and all of its destinations are ready to receive data. This allows many instructions throughout the program to be executed simultaneously.

The data in a data flow program can be modeled by "tokens" that reside on the arcs of the graph. Each arc may contain at most one token. The execution rule for most instructions is as follows:

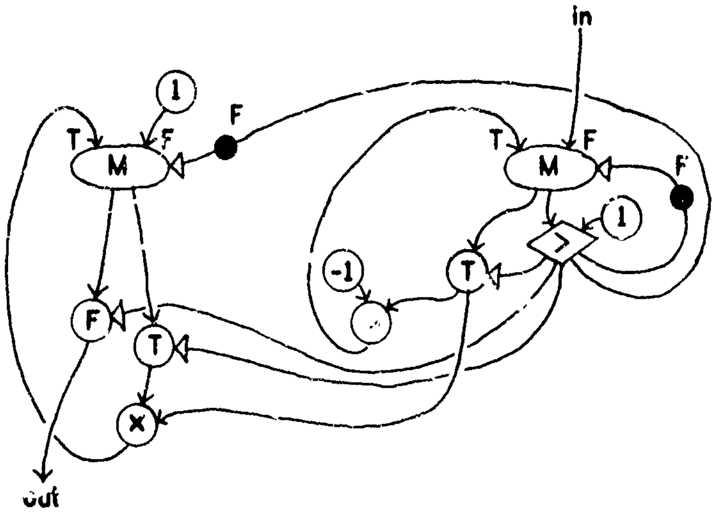
> An instruction (other than a merge or gate) is ready for execution whenever all of its input arcs contain tokens and all of its output arcs are empty. When an instruction is executed, the tokens on the input arcs are absorbed. The function denoted by the instruction is computed, using the values in the absorbed tokens as input data. A token containing the function value is placed on each output arc.

There are a number of ways of handling decisions and iteration control. Perhaps the simplest is the use of special instructions M, T, and F. These receive a boolean value on one input (the "control" input) and use it to control the passage of data from another input. Their execution rules are as follows:

> The M (merge) has a control input and two data inputs labelled "T" and "F". To be ready for execution, there must be a boolean token on the arc leading to its control input. Furthermore, the arc leading to whichever of its T or F input matches that boolean token must have a token, and all output arcs must be empty. When it is executed, the control token and the data token at the input indicated by the control token are absorbed. Copies of the token at the selected data input are placed on each output arc. Input tokens are not required at the non-selected data input, and if any are present they are not absorbed.
>
> The T (true gate) and F (false gate) instructions have a control input and a data input. They are ready for execution whenever both input arcs contain tokens and all output arcs are empty. When they are executed, the inputs are absorbed. If the control input matches the name of the instruction, copies of the data input are placed on the output arcs. If not, no tokens are placed on the output arcs.

Constants can be generated through the use of functions of no arguments. An instruction to perform such a function has no input arcs, so, in accordance with the execution rule, it places tokens on its output arc as fast as they are removed.

Here is an example of a data flow schema to compute the factorial function:



Boolean inputs to M, T, and F instructions are drawn as open arrows. Tokens existing in the initial configuration of the program are drawn as filled-in circles.

The behavior of a data flow schema under the execution rules has a very important property - it is determinate. This means that the output of the program is determined only by the input, and is independent of the timing of instruction executions. All runs of such a program with the same data will yield the same results. Determinacy follows from the facts that

(1) Each instruction produces a result which is a function only of the values of its input tokens, that is, each node of the schema is determinate.

(2) The value of a token does not change in any way while it resides on an arc.

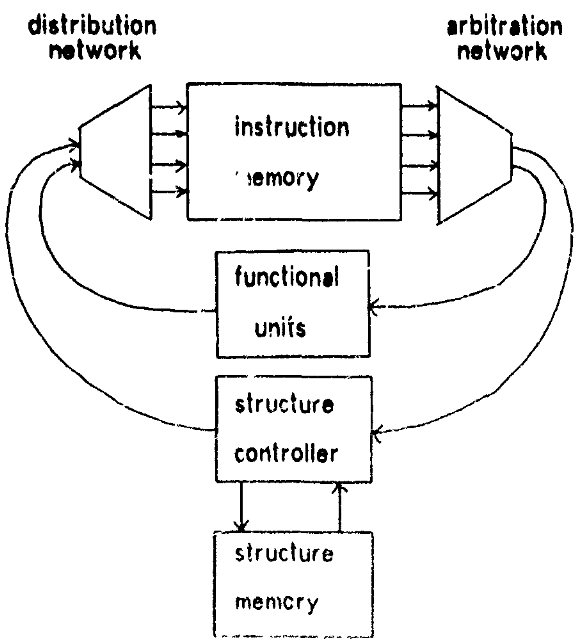
(3) The execution rules, and fact (2) above, qualify the schema as a valid interconnection of autonomous communicating systems.

It is an established result that such an interconnection of determinate systems is determinate [1] [14].

1.0.1 DATA FLOW COMPUTER ARCHITECTURE

The memory system and structure processor that are the subject of this thesis are intended to be part of a computer of the type described by Dennis and Misunas [6] [7]. Such a computer is composed of units which use packet communication [8] for transfer of data. The only means of data transmission among these units is the transmission of fixed size messages called packets. There is no clock or synchronizing information.

The four main parts of the data flow computer are the instruction memory, arbitration network, functional units, and distribution network. For structure processing, the structure controller and structure memory are added.



To execute a data flow program, its schema is encoded into the instruction memory. Each cell of the memory contains one instruction of the schema. At the time the program is loaded, each [illegible] structure operation, etc.) and the address of its destinations. The latter are the cells to which outgoing arcs point. The instruction cells also have receiver registers to contain incoming "tokens". When all necessary receiver registers become full, an instruction cell emits an operation packet, consisting of its operation code, the data from the receiver registers, and the destination addresses.

Any given program has a great number of instruction cells, each sending operation packets only occasionally. These streams of packets are merged by the arbitration network into a small number of dense streams. The packets coming out of the arbitration network are sorted according to operation code and sent to the appropriate functional units. In the case of structure processing instructions, they are sent to the structure controller. The functional units or structure controller perform the indicated operation and form, for each destination, a result packet consisting of the destination address and a copy of the actual result. The result packets go to the distribution network, where they are sorted by address and sent to the appropriate receiver register of the appropriate instruction cell. (The destination address includes the receiver number.) If the instruction is a structure operation, the structure controller may send numerous command packets to the memory and receive result packets back during the course of its computation.

The preceding description does not quite implement the execution rule: An instruction cell should wait until its "output arcs", that is, the receivers of its destinations, are empty before issuing an operation packet. There is no way for an instruction cell to "see" its destinations' receivers. The problem is remedied by using, where necessary, acknowledgment tokens sent from a cell's destinations to the cell itself. The acknowledges are treated like invisible arguments, except that they contain no data. When a cell is executed, it may send result packets to some destinations and acknowledges to others. A cell is not ready to be executed until it has received all necessary real arguments and all necessary acknowledges. Acknowledges are placed in the program where necessary to ensure that, when a cell has received all arguments and acknowledges, its destinations' receiver registers will be empty. These acknowledges should not be confused with the packet acknowledges to be developed later.

A constant need not be implemented as a separate node of the data flow schema. It can simply be loaded into the receiver register of the instruction cell that uses it, and marked in such a way that the instruction cell knows that that register is always full.

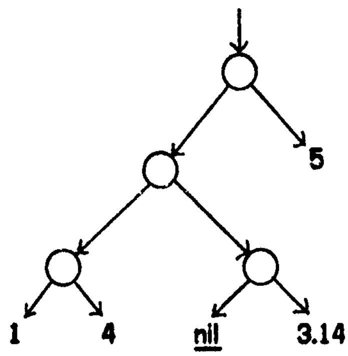
An additional part of the data flow computer, not shown in the preceding diagram, is the host computer. This is a computer of conventional design, which has access to the memory units and control functions of the data flow computer. It is used for diagnostic testing and for initial loading of the instruction memory and structure memory. It does not participate in the actual data flow computation.

1.1 DATA STRUCTURES

In order to handle arrays and data structures in a data flow computer, it is in most cases necessary to allow single tokens to have entire structures as their values. (Some programs which use arrays of fixed size, such as Fourier transforms and other signal processing algorithms, can make do with arrays of instructions with one token on each arc. However, this approach is impractical for very large arrays or for dynamic structures.) For this reason, we propose a data structure facility that allows tokens to have structure values. The simplest type of structure that permits full generality is the binary tree, which is recursively defined: a binary tree is an elementary "object" from some set, or is a concatenation of two binary trees. Such trees form the basis for the programming language LISP. [4] [13] For definiteness, the structures used in a data flow computer will be assumed to be binary trees.

The "elementary objects" are all data values other than structures that the computer can handle, plus the special object nil. Elementary objects thus might include integers, boolean values, reals, etc.

The principal operation on a data structure is selection. A simple selection takes a structure and a single bit. If the structure is elementary and not nil, the result of the selection is undefined. If the structure is nil, the result is nil. Otherwise, the structure is the concatenation of two structures, and the result of the selection is the first or second of these if the bit is zero or one respectively. A compound selection takes a structure and a string of bits, and gives the result of applying simple selections repeatedly, using the bits in sequence. The bit string is called the selector. Let S be the following structure:



SELECT[S, '1'] = 5   (a simple selection)

SELECT[S, '001'] = SELECT[SELECT[SELECT[S, '0'], '0'], '1'] = 4   (a compound selection)

The true "meaning" or "value" of a structure can be defined to be the set of ordered pairs of selectors that yield elementary values other than nil, along with those values. Thus the structure S denotes the set

{ <'000', 1>, <'001', 4>, <'011', 3.14>, <'1', 5> }

Nil simply denotes a substructure with no elementary items at all.

Using this definition of the meaning of a structure, there is a structure corresponding to any finite set of ordered pairs of selectors and elementary values (excluding nil) such that no selector in the set is an initial substring of another. The structure nil denotes the empty set.

SELECT[struc, sel] =

The elementary value v if struc contains the pair <sel, v>

Undefined if <s, v> ∈ struc where s is a proper initial substring of sel

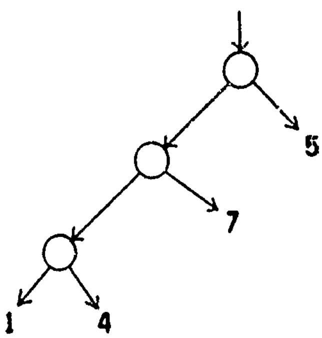
The structure { <s, v> | <sel•s, v> ∈ struc } otherwise

Structures can be built with the <u>append</u> operation. APPEND places a given object (structure or elementary value) onto a given structure with a given selector, removing whatever substructure previously existed there. In the set-theoretic model,

$$\text{APPEND}[\text{struc}, \text{new-val}, \text{sel}] =$$

$$\begin{cases} (\text{struc} - \{ \langle s, v \rangle \mid \text{one of sel or } s \text{ is an initial substring of the other}\}) \cup \{ \langle \text{sel}, \text{new-val} \rangle \} \\ \qquad \text{if new-val is elementary.} \\ (\text{struc} - \{ \langle s, v \rangle \mid \text{one of sel or } s \text{ is an initial substring of the other}\}) \cup \\ \qquad \{ \langle \text{sel} \cdot s, v \rangle \mid \langle s, v \rangle \in \text{new-val}\} \quad \text{if new-val is a structure, including } \underline{\text{nil}}. \end{cases}$$

Letting S be the structure defined previously, APPEND[S, 7, '01'] is



The substructure containing <u>nil</u> and 3.14 disappears.

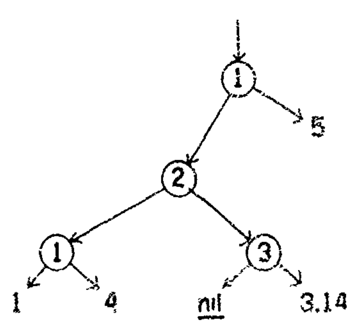
1.1.1 REPRESENTATION IN MEMORY

Structure can be implemented on a data flow computer in the same way that they are commonly implemented on ordinary computers - as linked lists of "cells" in a memory. An elementary object is represented by the object itself. A concatenation is represented by the address in memory of a cell containing the representations of the two substructures. In either case, a structure is represented by a small amount of information. The huge amount of information that constitutes the structure itself lies inside the memory, and the representation is merely a pointer to this. The operation of selection is quite simple. Cells are read from memory and the appropriate halves of the data used, under control of the selection bits.

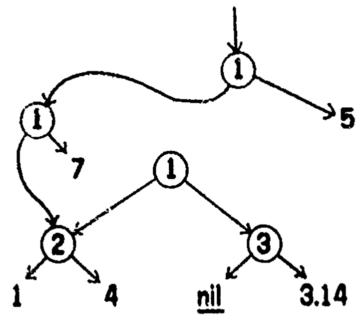
1.1.2 SHARING

Such an implementation leads to the possibility of a single structure in memory being shared (or partly shared) by several parts of the computation. In a data flow computer, two tokens might have the same pointer as their value. This is of course very desirable for economical memory use, but it makes the APPEND operation difficult. The problem is that modification of pointers inside the memory can change the value of structures other than the intended one, if structures have parts in common. In many programming languages, this is considered a reasonable and even desirable effect. For example, the LISP language has instructions to modify existing structures. In a data flow computer, however, this cannot be permitted for reasons of determinacy. In order for a data flow computer to be determinate, the meaning (in the set-theoretic sense given previously) of a token bearing a structure value must not change while that token resides on an arc. Since other instructions, including APPEND's, can be executed while a token resides on an arc, APPEND must never change any substructures that are shared with other structures.

In the proposed structure processing facility, each cell has a reference count which makes it easy to tell what substructures are shared. Whenever the APPEND processor is tempted to modify a cell that is shared with another structure, it makes a copy of the cell and modifies the copy instead. For example, if S is a pointer to the following structure in memory:



where the number in each node is the reference count, APPEND[S, 7, '01'] yields



The node that originally had a reference count of two may not be modified, so a copy is made, and its reference count is therefore reduced to one. The structure controller to be described in the next section will perform these tasks.

1.2 THE STRUCTURE CONTROLLER

In this section we will outline the behavior of a processing mechanism that uses the structure memory to provide a structure facility for the data flow computer. The basic behavior of the structure controller is that it receives operation packets from the arbitration network and delivers result packets to the distribution network. It holds the state information for structure operations in progress, and performs memory operations by sending packets to the memory and receiving packets in return.

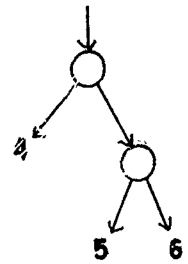
The purpose of this section is to show how the structure controller will use the memory, rather than to give a detailed specification for the structure controller. Therefore, a number of design decisions will be made arbitrarily. For the most part, the requirements of the structure memory are independent of these decisions. For example, the memory design would not change if ternary trees were used instead of binary ones.

Some aspects of the design of the structure controller will be considered in more detail in section 5.

1.2.1 DATA FORMAT

The memory space is divided into "words" or "cells", each of which holds one node of a structure. Since the memory is used for the storage of binary trees, the words representing nonterminal nodes contain two pointers to other nodes. The convention will be made that all words of the memory will be divided into halves, called the left half and the right half. Each half has an "elem" bit bit indicates whether it contains an elementary item (terminal node) or a pointer to another word in the memory. If the bit is 1, the half word contains an elementary value. The interpretation of that half word is then the exclusive responsibility of the rest of the computer, unless it is nil. The structure controller treats any elementary value other than nil simply as a collection of bits. Any type information (integer, floating point number, character, etc.) must be encoded into the half word along with the data.

The structure graphically represented as follows:



might be realized by address 102 in the following memory configuration:

location 102

1	4	0	107

location 107

1	5	1	6

The bit at the left end of each half word is the "elem" bit

(A different convention could be used, in which each elementary value takes an entire word instead of half a word. The two conventions are equally powerful, and differ only slightly in execution. The "half word" convention will be used for definiteness.)

1.2.2 MEMORY MANAGEMENT AND GARBAGE COLLECTION

All words of memory that are not part of a structure are kept in a collection of <u>free storage lists</u>. (There are several such lists, rather than one, in order to maintain a high rate of processing. This point will be discussed in section 5.0.5.) Whenever the structure controller needs a word in order to create a node, it takes it from one of the lists. Whenever a node is destroyed, that is, all pointers to it disappear, the word containing it is returned to a free storage list.

Each node of a structure has a reference count, which is the number of pointers to that node that exist, whether in other nodes or in the rest of the computer. (The latter includes operands waiting in instruction cells and packets in transit through the arbitration and distribution networks.) The structure controller increases or decreases the reference count of each node as pointers to it are created and destroyed. When the reference count is decreased to zero, the node disappears, so it is returned to a free storage list. Whenever this happens, any pointers that the node contained disappear, and so the reference counts of the nodes pointed to must be decreased.

The choice of a reference count strategy for memory management instead of the "mark and scan" method commonly used in LISP systems was made for three reasons:

(1) The mark and scan method requires a garbage collection operation which must find every reference to every structure. Since references exist in packets in transit, it would be necessary to stop the entire computation and wait until all packets stop moving before a garbage collection commences.

(2) The reference count is needed anyway in order to implement the copying rule efficiently. Whenever the structure controller needs to modify a node as part of an APPEND operation, it may do so safely if the reference count is one. If not, the node must be copied.

(3) The objections to the reference count method in many list processing systems, that it is difficult to recover circular lists, does not apply here. Because of the copy rule, circular lists are never created.

1.2.3 THE STRUCTURE OPERATIONS

The structure controller to be proposed implements the following program level operations:

SELECT(structure, selector) - The selector is a bit string of definite length. The structure is traced under control of the bits in the selector, starting with the leftmost bit. A zero bit selects the left offspring and a one bit selects the right. The item at the selected point in the structure is returned, whether it is elementary or a substructure.

APPEND(structure, object, selector) - Returns a structure similar to the given one, but having the object at the place specified by the selector. Whatever was at that place in the original structure is absent in the result. The object may be elementary or a structure. Any part of the original structure that is shared with other parts of the computation is not modified. The controller copies part or all of the original structure as necessary to be sure that this is the case.

The structure controller recognizes the special constant nil which, while elementary, is also the structure with no selectors. Nil is used as a terminal node of a structure to indicate that there are no objects beyond that point. Any part of a structure may be deleted simply by using the APPEND operation to replace it with nil, and a structure may be created by appending something to nil. It is assumed that the constant nil is explicitly available to the programmer for these purposes. The controller optimizes all structures, replacing with nil any substructure all of whose terminal nodes are nil.

There are two more operations performed implicitly by the controller. If any operation returning a structure value specifies more than one destination, the reference count of the result must be appropriately increased. Also, if any operation discards a structure value, the reference count must be decreased. It follows that the conditional operations such as true and false actors must be executed by a structure controller if the objects being switched are structures.

1.2.4 THE MEMORY OPERATIONS

The structure controller communicates with the memory by sending command packets and receiving result packets. These packets are given names describing the operation to be performed.

To read a word of memory, a FET ("fetch") packet is sent, giving the address. The memory returns a LOAD packet with the data. Between the FET and the corresponding LOAD, many other packets might be sent and received. This is a consequence of the parallelism of the data flow computer: just as with the other functional units, the rate at which structure operations are performed can be increased by allowing many operations to be in progress simultaneously. This concurrency is made possible by the use of packet communication at the memory interface. The FET packet that begins an operation and the LOAD packet that ends it are distinct events and might be separated by a great number of other packet transmissions and receptions. Each LOAD packet is identified with the FET packet that caused it by means of the "tag", to be described later.

Each LOAD packet contains the address of the word and its reference count, as well as the data. The address is probably not used by the structure controller, but is included as part of the specification of the memory module because it is needed by the cache mechanism to be described in section 3.2. The structure controller uses the reference count in order to tell when a node may be written on without being copied (if count = 1) and when a node should be destroyed (if count = 0).

To increase or decrease the reference count of a word, the FET^+ or FET^- packets, respectively, are sent. These are similar to FET, except that the reference count is first modified. The memory replies to them with $LOAD^+$ or $LOAD^-$ packets which are similar to LOAD packets. In some cases the structure controller does not use the data in a $LOAD^+$ or $LOAD^-$ packet, but it does not really cost anything for the memory to send it.

To write on a word of memory, the structure controller sends an UPD ("update") packet giving the address, data, and reference count. The reference count is presumably one, but the specification of the memory module allows an arbitrary count to be given. (In an actual implementation of a structure controller and memory, unnecessary fields would be omitted where possible, so that the controller would not send a reference count in UPD packets or receive an address in LOAD, LOAD$^+$, or LOAD$^-$ packets.) The memory sends no reply to an UPD packet.

There is another command that the memory recognizes. The CLR packet waits until all pending operations on the given word are complete, and then returns a DONE packet. It is not used by the structure controller at all, but is required for operation of the cache.

1.2.5 THE TAG FIELD

Every FET, FET$^+$, or FET$^-$ packet has a field called the "tag" field that constitutes a reminder from the structure controller to itself, telling it what to do with the result of the operation. The tag field of a command packet is returned unchanged in the result packet.

Consider the case of a simple SELECT instruction. When the instruction cell fires, an operation packet goes to the structure controller containing the operation code, the structure, the selector, and the addresses of the the instruction cells which are to receive the result. There might typically be three such destination addresses, each about 20 bits long. The structure controller can simply make them the tag field of the "fetch" command to the memory, and then use them when they come back in the result packet. In the case of more complicated structure operations, such as APPEND's with compound selectors, there is a large amount of state information that must be remembered through the many memory transactions that make up the structure operation. In addition to the destination addresses, there is the datum to be appended, the structure to be ultimately returned, the remaining selector bits, and a few pointers. The total amount of such data typically might be 200 bits or more.

There are two ways of handling this information. One method is to include all of it in the tag field of commands to the memory, so the structure controller doesn't need to store any information about the state of ongoing structure operations. When the result packet comes back from the memory, the structure controller looks at the entire packet including the tag field, decides what to do next, and produces a new packet to send back to the memory. This method (the "memoryless structure controller" method) is efficient, but it requires an extremely wide data path for all memory transactions, and it gives rise to very difficult problems of avoiding deadlocks.

A second method is to store all of the state information in the structure controller. This requires that the controller have a memory with a capacity of 200 bits or more for every structure operation that can be in progress at one time. In this case only the address of the block of memory in which the state information is stored must be put in the tag field. If 256 simultaneous structure operations are allowed, the tag field only needs to be 8 bits.

In either case, commands to the memory contain a tag field. The memory echoes the tag back to the controller in the result packet.

1.2.6 THE DATA AND REFERENCE COUNT FIELDS

The contents of each memory word consists of a data field and a reference count field. The data field is further divided into two pointer fields, leaf-node indicator bits, perhaps a bit to indicate that the cell is on the free storage list, and perhaps type indicator fields for elementary values. All of these are significant only to the structure controller, and are irrelevant to the memory. The memory can simply consider the data to be a homogeneous field. In practice, it might be about 40 to 80 bits long.

From the memory's standpoint, the reference count is simply part of the data associated with each word. In some transient cases it might become negative in some parts of the memory system, although the structure controller will never see a negative reference count. In a typical realization, the reference count field might be about 8 to 15 bits long.

Incoming and outgoing packets that read or write a word of memory have data and reference count fields that correspond precisely to the fields in memory.

2.0 SPECIFICATIONS OF PACKET SYSTEMS

In this section we will develop methods by which one may describe how a hardware system using the packet communication principle is constructed, how such a system behaves, and how one may prove that a system constructed in a certain way behaves in a certain way. Examples will be given of simple systems that illustrate some of the important points of the design method.

2.0.1 FUNCTIONAL SPECIFICATIONS

Because of the restricted way in which packet communication systems interact with their environment, it is easy to describe how such a system might behave. Since the only interaction is through packets, a system's behavior is completely known if it is known what packets it will transmit in response to whatever packets are sent to it. One other piece of information that might be available, but that we reject, is the time when a packet is transmitted. It is impermissible for a system to be described as, for example, transmitting the result of a computation between 1 and 1.5 microseconds after it receives the data. The only requirement is that it eventually produce the result. (This is not to say that speed is unimportant. Like any other computer, a data flow computer is designed with operating speed in mind. The conditions for correct behavior are independent of speed, however. If any component of a data flow computer is replaced with one that operates at a different speed, the computer will continue to function correctly.)

Since a module of a packet communication system may retain internal state information (though many useful modules do not), the "result" packets that it transmits may depend not just on individual input packets, but on the entire history of input packets. All packets that pass through a given input or output port have a definite order among themselves. The ordered sequence of packets that have passed through a port from the time that the system was started up until a given instant is the history of the port at that instant. A history of a port will be written by listing the packets in parentheses, separated by semicolons.

There is a partial order on histories: $X \leq Y$ if X is an initial subsequence of Y. For example:

$$(1 ; 3 ; 4) \leq (1 ; 3 ; 4 ; 7)$$

but (1 ; 2 ; 4) and (1 ; 3 ; 4) do not satisfy this relation in either order.

Since histories only grow longer as time progresses and symbols already in a history never change, a history at a later instant is always greater than or equal to a history at an earlier instant.

The length of port history X is denoted $|X|$. The individual packets of X are $X_1, X_2 \ldots X_{|X|}$.

There is no defined time order among packet arrivals on different ports, so it is useless to represent them as a single sequence. Instead, a history array is used, which is a collection of histories, one per port. The partial order on histories can be extended to arrays: $A \geq B$ if each history of A is greater than or equal to the corresponding history of B. Like histories, history arrays increase as time progresses.

The description of how a system is expected to behave is quite simple. It is a description, for every input history array, of what output history array the system will eventually produce. "Eventually" means in finite time for finite histories. For infinite histories, it means that, for any K, the first K packets will be produced in finite time. This is because a system which is expected to have an infinite output history cannot ever transmit its entire output in finite time.

A description of the dependence of output history arrays on input arrays is called a functional specification. It is a description of how a system is expected to behave. The major problems in the field of packet communication systems are proving that a system built in a certain way obeys a certain functional specification, and proving that the interconnection of systems known to obey certain functional specifications obeys some other functional specification.

If, for any input array, the functional specification states that there is only one possible output array, the system is determinate (sometimes called functional, but that term will not be used here). In that case there is a function, say f, mapping input arrays to output arrays, such that, if input X (and no more) is delivered to the system, f(X) will eventually be produced. If further input is then given, the input history is Y with $Y \geq X$, and output history f(Y) will be produced, Since the system cannot retract any of its previous output, $f(Y) \geq f(X)$. From this it is easy to see that f is monotonic in that:

$$X \geq Y \Rightarrow f(Y) \geq f(X)$$

If there is more than one legal response to a given input array, the system is nondeterminate. In that case a function is also used to define the functional specification, but f(X) is the set of all legal output history arrays. Functions defining the specifications of nondeterminate systems also obey a sort of monotonicity property, which will be given later.

It is possible for an interconnection of nondeterminate systems to be determinate. For example, a data flow computer is determinate even though its arbitration network is not. An interconnection of determinate systems is always determinate, and its function can be computed explicitly from the functions of the components [1].

2.0.2 DESCRIPTIVE SPECIFICATIONS

Since a major task of the system designer is to demonstrate that a system built in a certain way obeys certain functional specifications, it is necessary to describe in a reasonably formal way how a system is built. A wiring diagram is one formalism, but it is far too rigid and implementation-dependent. A higher level method is needed. When a system is assembled from components, all using the packet communication principle, it is of course easy to describe the interconnection, telling what ports of the various systems are connected to each other. For systems that cannot be so decomposed, the descriptive specification will be given in terms of a program written in an extremely informal ALGOL-like language. This language is a subset of the Architecture Description Language [10] which is under development.

In the language we will use for giving descriptive specifications, packets will look like data records with a title and one or more data fields, for example: "WRITE(3, 7)". This format is purely cosmetic. In the actual hardware implementation, a packet is nothing but a collection of bits. The fields are simply divisions of these bits into subsets that the sender and receiver both agree upon. The titles are just encodings of another field.

2.0.3 AN EXAMPLE OF A DETERMINATE MEMORY

A functional and descriptive specification of a system called MEM will now be given. MEM is a random access memory with an input port IN and an output port OUT. Two types of packets may be delivered to it:

WRITE(addr, data) writes the data into the given address
READ(addr) fetches the data from the given address

The "addr" and "data" fields contain numbers that range over some finite and fixed spaces. There is one output packet type:

RTR(addr, data)
(RTR stands for "retrieve")

Every READ packet delivered to MEM results in transmission of a RTR packet bearing the address and the current contents of the memory. Every WRITE packet stores its data in the memory and returns no result packet. The initial contents of each address of the memory is zero.

For a given input history, the contents of the memory may be easily determined. The contents of each word is simply the data field of the last WRITE packet having that address, or zero if there is no such packet. The function f_{MEM} realized by this memory is:

f_{MEM}

If X = input history and Y = output history,

$f_{MEM}(X) = Y$ where

$|Y|$ = the number of occurrences of READ(--) in X

Y_i = RTR(addr, data) if the i^{th} READ(--) in X is READ(addr) and the last WRITE(addr,--) in X before that READ is WRITE(addr, data), if there is such a WRITE

RTR(addr, 0) if the i^{th} READ(--) in X is READ(addr) but there is no WRITE(addr,--) before it

Notation: WRITE(addr,--) means any WRITE packet having the specified addr field and anything at all in the data field.

A functional specification of MEM simply consists of stating that MEM realizes f_{MEM}, that is, that is the input history X is presented to it, it will eventually transmit output history $f_{MEM}(X)$.

This specification says nothing explicit about the states of MEM. This is a basic property of the history function approach to system specification - even for a device whose purpose is to have states, such as a memory, the specification does not mention the states. Of course, the memory does have states, and the state is a function of the input history. Since the input history records all of the information that has ever gone into the system, it contains enough information to determine the state.

We now show how the system MEM may be built: The system uses a re random access memory, with a capacity of one word for each possible value of the "addr" field of incoming packets. We choose some obvious correspondence between the values of the "addr" field and word addresses. Each word can contain any of the possible values of the "data" field of incoming WRITE packets. We choose some obvious correspondence here also. The memory is initialized with all words containing zero.

The algorit of the implementation of MEM is as follows: If a packet WRITE(addr, data) is received, the data field is written into memory at the word address given by the addr field. If a packet READ(addr) is received, the word at the appropriate address is nondestructively read, and a packet RTR(addr, data) containing the data fetched from memory, is returned

This system may be implemented by the program which follows. "Memory" is an array which represents the actual memory.

```
    process starts at A
    input port IN
    output port OUT
    var command, addr, data
    array memory init 0

| wait for input

    A:    until packet is available at IN do;
          command := packet from port IN;

| analyze input packet

          if command = READ(--) then
             let command = READ(addr);
             send RTR(addr, memory(addr)) at port OUT
          else
```

```
    let command = WRITE(addr, data);
    memory(addr) := data;

goto A
```

Notes:

(1) The statements for receiving and transmitting packets are excessively primitive. Slightly improved versions will be presented later.

(2) The expression RTR(addr,data) means "a RTR packet whose fields are filled with the current values contained in addr and data".

(3) The "--" in conditionals has its usual meaning. "If packet = WRITE(3,--)" means "if packet is a WRITE packet whose first field is 3".

(4) The "let packet = pattern" statement is an assignment statement that sets the variables appearing in the pattern to have the values of the corresponding fields of the packet. "let thing = WRITE(addr,--)" means "if the type of thing is not WRITE, it is an error; otherwise set addr to the first field of thing and ignore the second field".

We now prove that this implementation satisfies the specification f_{MEM}. First, we need to show that the memory state equals the system state (as defined by the input history) under the following correspondence:

For all X, the contents of memory address X for a given input history is

zero if the input history contains no packets WRITE(X,--)
Y if the history does contain such packets, and the last is WRITE(X,Y)

Proof by induction on the length of the history at port IN. For length zero, all cells contain zero by initialization, and the history contains no WRITE packets at all. Otherwise assume true for any history of length K and prove it for K+1.

If IN_{K+1} = READ(--), nothing was written into memory between receipt of IN_K and IN_{K+1}, so the memory state did not change. The existence of WRITE(--,--) packets did not change either.

If IN_{K+1} = WRITE(addr, data), no memory cell other than addr changed, and the existance of WRITE(X,--) packets did not change for X ≠ addr. The contents of memory cell addr is now data, and the last WRITE(addr,--) in the history is now obviously WRITE(addr, data).

Next, we prove correctness of the implementation. If the input history = X, we will show that $f_{MEM}(X)$ will appear at the output. This proof is also by induction. If $|X| = 0$, $f_{MEM} = \epsilon$. But the implementation specifies no output except in response to input. Now suppose $X' = x_1x_2 \ldots x_Nx_{N+1}$. Let $X = x_1x_2 \ldots x_N$. By induction, $f_{MEM}(X)$ appeared at the output when X was the input history. When x_{N+1} arrived, the system transmitted no output if x_{N+1} was a WRITE, and transmitted RTR(addr, memory(addr)) if x_{N+1} was READ(addr). Therefore the response to X' is

$f_{MEM}(X)$ concatenated with

$$\begin{cases} \epsilon \text{ if } x_{N+1} = \text{WRITE(--,--)} \\ \text{RTR(addr, memory(addr)) if } x_{N+1} = \text{READ(addr), where the memory} \\ \qquad\qquad \text{state is that left by X} \end{cases}$$

Now $|f_{MEM}(X')| = |f_{MEM}(X)| + 1$ if x_{N+1} is READ(--), which is the length of the response to X'.

Also, if x_{N+1} = WRITE(--,--), $f_{MEM}(X') = f_{MEM}(X)$, and if x_{N+1} = READ(addr), $f_{MEM}(X')$ = $f_{MEM}(X)$ concatenated with RTR(addr, z), where z = the data field of the last WRITE(addr,--)

packet, or zero if there is none. This is just the contents of memory word addr.

The response to X' is therefore $f_{MEM}(X')$.

This system has a few simplifying properties that a general system of the sort to be used in the packet memory system can't have:

1) It is determinate.

2) Its behavior is defined for all possible input histories, that is, there are no illegal inputs.

3) It operates infinitely fast, that is, it is impossible for input commands to come too fast for it to handle. (Note that the above proof says "when x_{N+1} arrived, the system transmitted ...")

2.1 NONDETERMINACY

Nondeterminate systems can take a wide variety of forms, and the problem of formalizing the behavior of all nondeterminate systems is far too complex to be treated in this thesis. Only the types of nondeterminacy that arise in connection with the structure facility for the data flow machine will be treated.

The principal type of nondeterminacy that will arise in packet memory systems is the removal of the requirement that the RTR packets be returned in the same order as the READ packets that gave rise to them. For example, the input history

WRITE(1,11) ; WRITE(2,22) ; READ(1) ; READ(2)   could result in

RTR(1,11) ; RTR(2,22)   or in   RTR(2,22) ; RTR(1,11)

The system MEM is too simple to display this sort of nondeterminacy. For example, MEM would return RTR(1,11) as soon as it received the first READ packet. It would not yet "know" that it was about to receive a second READ packet which would give it the option of producing its output packets in either of two orders. Later, we will exhibit implementations of systems which can meaningfully take advantage of this nondeterminacy. For now, we will just have to accept that such implementations (that is, descriptive specifications) exist, and examine the form that the functional specification for such a system might take.

2.1.1 FUNCTIONAL SPECIFICATIONS OF NONDETERMINATE SYSTEMS

A nondeterminate system can give any of several legal output histories in response to a given input history. The "function" defining the system's behavior is therefore multiple valued. One way to handle this situation is to treat the behavior of a system as being defined by a relation instead of a function. The method to be used here, which is completely equivalent, is to use functions whose values are sets of output histories. For example, in the system f_{NDMEM} that we are developing,

$$f_{NDMEM}(WRITE(1,11)\ ;\ WRITE(2,22)\ ;\ READ(1)\ ;\ READ(2)) =$$

$$\{\ (\ RTR(1,11)\ ;\ RTR(2,22)\ )\ ,\ (\ RTR(2,22)\ ;\ RTR(1,11)\ )\ \}$$

The situation may arise that f(X) is empty for some X. This means that X is not a valid input history, and the behavior of the system is undefined. This is different from the situation in which an illegal input gives rise to a well-defined "error" response (packet) from the system. An "error" packet is certainly more desirable than saying the system behavior is undefined, but some situations, such as receiving acknowledges for packets that were not sent, are so pathological they must simply be assumed not to occur. Furthermore, at some levels of detail in the description of a system, it is convenient to ignore error conditions if one can prove that they won't occur when the system is functioning properly.

A functional description of a nondeterminate system is therefore a definition of a function which maps input histories into sets of output histories. It is usually most convenient to describe it as a predicate defining which histories are in f(X) for a given X, and that predicate is often the logical AND of a number of other predicates, so the functional description looks like:

Y is in f(X) if

$P_1(X,Y)$ and
$P_2(X,Y)$ etc.

The rule for realization of a function is as follows: A system realizes f if, given input history X with f(X) nonempty, it will eventually produce some output history in f(X).

The multiple valued functions realized by nondeterminate systems must obey a monotonicity property as follows:

NONDETERMINATE MONOTONICITY (ND-MONOTONICITY)

If Q and P are input histories and $Q \geq P$, then for any output history X in f(P), if f(Q) is nonempty there is a history Y in f(Q) with $Y \geq X$.

Roughly speaking, this means that receipt of a legal input symbol will never make the system unable to proceed legally. The purpose of the qualification "if f(Q) is nonempty" is to allow for the possibility that an illegal input packet might make the system unable to proceed.

We can now give the functional specification for the nondeterminate memory NDMEM, which can arbitrarily mix RTR packets for different addresses.

f_{NDMEM}

If X = input history and Y = output history,
Y is in f_{NDMEM}(X) if

(1) Y consists only of packets RTR(--,--), and
(2) For all addr, the number of READ(addr)'s in X = the number of RTR(addr,--)'s in Y, and
(3) For all addr and K, the K^{th} RTR(addr,--) in Y, if it exists, is RTR(addr,val) where last WRITE(addr,--) in X before K^{th} READ(addr) in X is WRITE(addr,val) if such a WRITE(addr,--) exists, or val = 0 if no WRITE(addr,--) exists before the K^{th} READ(addr) in X

The system NDMEM has the property that the data returned in a RTR packet is the data in the memory (that is, the data in the most recent WRITE command addressing that cell) at the instant of the READ command corresponding to the RTR. At the instant the RTR packet is sent out, another WRITE command might have already been received, but that WRITE will have no effect on this RTR packet.

Example

input: WRITE(A,1) READ(A) WRITE(A,2) READ(A)

output: RTR(A,1) RTR(A,2)

$\longrightarrow$ time

At the instant the first RTR packet was returned, a WRITE command changing the data from 1 to 2 had already beeen given, but the function f_{NDMEM} requires that the value 1 be returned.

Here is a rough outline of an implementation of a system that realizes f_{NDMEM} :

SYSTEM #1 (realizing f_{NDMEM})

(1) When a WRITE command comes in, write on the word of memory instantly.

(2) When a READ command comes in, fetch the word from memory instantly, form a RTR message, and put it into a buffer or queue.

(3) Take messages out of the buffer and return them as output packets at any time and in any order, subject to the restrictions that:

(a) every packet in the buffer is eventually removed,

(b) whenever a packet is removed, it must be the oldest in the buffer among those with its word address (that is, the buffer is first-in-first-out (FIFO) with respect to each address).

The implementation given above still requires that operations on the memory be instantaneous, so it is not very useful because it doesn't take advantage of the delay between a READ packet and the RTR packet that results. The data in the RTR packet must be the contents of the memory word at the instant the READ/RTR interval begins. We would like the system to be able to use the value of the memory word at any instant during the READ/RTR interval. Here is an example of a system that takes such liberty:

SYSTEM #2 (purported realization of f_{NDMEM})

(1) When a WRITE command comes in, write the word of memory instantly.

(2) When a READ command comes in, put the message READ(addr) in the Pending Read Buffer (PRB).

(3) Take messages off the PRB at any time and subject to the same restrictions as before, namely that every message is eventually removed and the buffer is FIFO on each address. When the message READ(addr) is taken from the Pending Read Buffer, fetch the data from memory and form a message RTR(addr,data). Send the latter to the Finished Read Buffer (FRB).

(4) Take messages off the FRB at any time and in any order subject to the same restrictions as before, form a RTR packet, and send it as output of the system.

This implementation does not realize f_{NDMEM}. In the packet timing graph after the definition of f_{NDMEM}, the first RTR packet might have value 1 or 2 if this implementation is used. (The second RTR packet will always have data value 2.)

We might like the system to take even more liberty, by performing memory writes, as well as reads, whenever it wishes. Such an implementation might be as follows:

System #3 (purported realization of f_{NDMEM})

(1) When a WRITE packet comes in, put the message WRITE(addr,data) on the Pending Write Buffer (PWB).

(2) Same as (2) in System #2.

(3) Take messages off the PWB subject to the same restrictions as before, and write the data into memory.

(4) Same as (3) in System #2, except that there is an additional

restriction that no message may be taken from the PRB if a message addressing that word is on the PWB.

(5) Same as (4) in System #2.

This too fails to realize f_{NDMEM}. However, both System #2 and System #3 do realize f_{NDMEM} if no WRITE packet is ever sent to the system when any READ/RTR transactions are in progress on that word. That is, before a WRITE packet is sent, a RTR packet must have been received for every READ packet sent addressing that word. Fortunately, it is not difficult to guarantee that this requirement is met: it is simply a nondeterminate functional specification for the "rest of the world", which we will call the "user".

Definition: The user of a system is that to which the system connects, and is itself a system. The input ports of the user are the output ports of the given system, and vice-versa.

It would of course be totally useless to require that, in order for a realization of f_{NDMEM} to work, its user must realize a determinate functional specification. In fact, the user of a system should have as few restrictions on its behavior as possible. Such restrictions can generally be specified by requiring that the user realize some nondeterminate function, just as the system itself does. That is, the difference between system specifications and user specifications is nothing but a matter of degree of restrictiveness.

The requirement that NDMEM's user not send a WRITE command when any READ/RTR transactions are in progress can be met by requiring it to realize the following nondeterminate functional specification $f_{NDMEMUSER}$:

$f_{NDMEMUSER}$

If Y = input history of USER and X = output history,
(note the exchange of input and output so that X and Y
refer to the same packet streams in both the system and its user)

then X is in $f_{NDMEMUSER}(Y)$ if

(1) X consists only of packets READ(--) and WRITE(--,--)

(2) For all addr, for any WRITE(addr,--) in X, the number of READ(addr)'s preceding it in X is $\leq$ the number of RTR(addr,--)'s in Y

The function $f_{NDMEMUSER}$ is easily seen to be ND-monotonic. This is because the restrictions on the user's output X never become more stringent as Y increases. As Y increases, the proposition "the number of READ(addr)'s preceding it in X is $\leq$ the number of RTR(addr,--)'s in Y" never goes from true to false, so the set of legal arrays X does not decrease. (If the "$\leq$" had been replaced by "=", it would not be ND-monotonic.)

While system #3 does not by itself realize f_{NDMEM}, it does realize f_{NDMEM} if connected to a user that realizes f_{USER}. To prove this, the important step is to show that each READ(addr) packet generates a RTR packet containing data defined by the most recent WRITE(addr,--) packet proceding the given READ(addr) packet in the input stream.

Let t_0 = the instant when the READ(addr) packet comes in. There may be pending WRITE(addr,--) packets in the PWB at t_0. If there are none, the most recent WRITE(addr,--) packet in the input stream has already passed out of the PWB and into the memory unit, so its data is in memory word <u>addr.</u> If there are WRITE(addr,--) packets in the PWB at t_0, the most recently inserted packet there is the most recent WRITE(addr,--) packet in the input stream. Therefore, letting

$$D_{addr}(t) = \begin{cases} \text{the data in the youngest WRITE(addr,--) packet in the PWB at time } t \\ \quad \text{if there is such a packet} \\ \text{the contents of word } \underline{\text{addr}} \text{ in the memory unit if not,} \end{cases}$$

we must show that the data to be eventually returned in a RTR packet is $D_{addr}(t_0)$. Let t_1 = the instant when the READ(addr) packet leaves the PRB. First, we show that $D_{addr}(t)$ does not change from t_0 to t_1. Since the READ(addr) packet has entered the system, it has left the user. Since the corresponding RTR(addr,--) packet has not yet been generated by the system (and won't be until after t_1), it has not been received by the user. Therefore, there is a READ/RTR transaction pending on addr, so the user is not sending any WRITE(addr,--) packets. Therefore, whichever WRITE(addr,--) packet in the PWB is youngest will stay youngest as long as it stays in the PWB. So as long as there are any WRITE(addr,--) packets in the PWB, D_{addr} does not change. As long as there are no WRITE(addr,--) packets in the PWB, D_{addr} = the contents of memory, which doesn't change either, because only removal of a WRITE(addr,--) packet from the PWB can change the contents of memory word addr.

There can be no transitions from no WRITE(addr,--) packets in the PWB to one or more packets, because the user is not sending any. The remaining case to consider is the disappearance of the last WRITE(addr,--) packet from the PWB. This packet is clearly the youngest, so D_{addr}(just prior to disappearance) = the data in the packet. This data is written into memory by rule 3 of the implementation. D_{addr}(just after disappearance) = data written into memory = data in the packet that disappeared. Therefore $D_{addr}(t_0) = D_{addr}(t_1)$.

At time t_1, when the READ(addr) packet leaves the PRB, there are no WRITE(addr,--) packets in the PWB, by rule 4 of the implementation. Therefore $D_{addr}(t_0)$ = $D_{addr}(t_1)$ = contents of memory word addr at t_1. But when the READ(addr) packet is taken from the PRB, the memory word is read, and its data goes into a RTR(addr,--) packet in the FRB. That packet is therefore RTR(addr,$D_{addr}(t_0)$), and is the packet that will eventually be returned to the user.

This example demonstrates a general principle:

> Whether or not a given implementation of a system realizes a given function may depend on whether the system's user realizes some other specific function.

There is no way to get around this fact. There are systems that correctly realize useful functions (even completely determinate functions) when connected to systems that obey certain rules, but behave in a totally pathological way otherwise. Furthermore, the system often can't tell whether the user has broken the rules. In the case of system #3 above, the system would have been able to tell whether a WRITE(addr,--) packet came in while a READ/RTR transaction was pending on word addr, but in some cases the system has no way of knowing whether its user is misbehaving.

The structure controller and packet memory system for a data flow computer is such a system. Perhaps the most important example of the structure controller and memory's dependence on the behavior of their user is the reference count and garbage collection problem. The rules that the user (i.e. the data flow computer) must obey in order to assure correct reference accounting are as follows:

(1) No pointer to a structure may be duplicated without giving a command to increase the reference count.

(2) No command to decrease the reference count may be given unless a pointer is discarded.

These rules guarantee that the reference count for a node is at least as great as the number of pointers to the node contained anywhere in the computer. (Actually, the rules will be such that the reference count is exactly equal to the number of pointers to the node. However, the penalty for too high a reference count is simply that a useless structure fails to be reclaimed and wastes memory space.)

Now suppose the computer (that is, the structure controller's and memory's user) violates the rule and allows the reference count to become too small. Eventually the reference count may become zero while a pointer to the node still exists somewhere. When the count goes to zero, the memory system reclaims the node and puts it on the list of free nodes.

Two possibilities then arise. If an immediate attempt is made to use the "spurious" pointer to the cell, in a SELECT instruction for example, the structure controller will send a READ command to the memory. The memory will know that this is an illegal command, that is, that the user has violated its specification. It can then signal an appropriate error condition in order to prevent the computation from giving an incorrect result.

If, on the other hand, the cell is removed from the free storage list and used by the structure controller to build some new structure by the time the spurious pointer is used, there is no way the memory can tell that a violation has occurred. It has no choice but to process the spurious command in the normal way, which results in its referring to a structure which is completely different from what was intended.

This is not to say that the data flow computer has no way to check for errors in the handling of reference counts. Methods of doing so will be discussed in section 5.0.5.

2.1.2 MUTUAL CONSISTENCY OF FUNCTIONAL REALIZATIONS

Suppose a system realizes f_{SYS} contingent on its user realizing f_{USER}, which the user does if the original system realizes f_{SYS}. Does it follow that the realizations actually occur when the two systems are connected to each other? Is it possible that they could both violate their specifications, with each blaming the other? Let the systems be S and T. Each is the other's user.

If any violation does occur, there must be a first instant of violation. That is, there is an instant t_0 when it first becomes true that one system (say S) has an output history which does not legally follow from its input history. There is a delay, however slight (even if it is only the delay caused by propagation of electric currents through wires) in the behavior of S. Therefore S's output history at t_0 depends on T's output history slightly before t_0, at a time when T was not malfunctioning, so S cannot blame its malfunction on T. Even if S and T both malfunction at precisely the same instant, neither S nor T knows about the malfunction of the other at that instant, and so neither malfunction can be excused. It follows that, if both systems conditionally obey their functional specifications, they will obey their specifications in practice.

2.1.3 MONOTONICITY OF FUNCTIONAL SPECIFICATIONS OF THE USER

We now give an example of how not to define the functional specification of a user. Suppose the system MEM has destructive readout, so that it requires that the user rewrite any data that it reads. Suppose further that for some reason the same data must be rewritten, and that it must be done immediately, that is, no other transactions may take place at any address between the read and the rewrite. Here is an attempt at a functional specification for USER. Since USER doesn't know what data to write until it receives the RTR packet, we will require the rewrite to be a consequence of the RTR.

f_{USER}

Y = input to user, X = output from user

For all addr and i, if the i^{th} RTR(addr) exists in Y and is RTR(addr,data), then the i^{th} READ(addr) in X is immediately followed in X by WRITE(addr,data)

Unfortunately, this does not require the user to wait for the RTR packet after sending any READ, not sending any more packets until the RTR arrives. For example, the user might send

(READ(1) ; READ(2))

Until the RTR(1,data) packet comes back, the user has not broken any rules. When the RTR(1,data) does come back, the user will have retroactively broken the rules and be unable to do anything about it. Since we would like to simplify as much as possible the task of proving that systems obey functional specifications, we need to make the specifications reflect the types of decisions that systems make in practice. It doesn't make sense for a system to perform some operation or emit some result packet on the basis of an input packet not having arrived and not being about to arrive, so f_{USER}, as given above, is unreasonable.

The problem is that f_{USER} is not ND-monotonic. To see this, refer to the notation in the definition of ND-monotonicity and let

$$P = \epsilon \qquad Q = \mathrm{RTR}(1,\mathrm{data}) \qquad \text{[input histories]}$$

$$X = (\mathrm{READ}(1)\ ;\ \mathrm{READ}(2)) \qquad \text{[output history]}$$

Now $Q \geq P$, X is in $f_{USER}(P)$ and $f_{USER}(Q)$ is nonempty (containing, for example, READ(1) ; WRITE(1,data) ; READ(2)), but there is no history in $f_{USER}(Q)$ that is $\geq X$.

The correct specification for the user is:

If Y = input to user, X = output from user For all addr and i, the i^{th} READ(addr) in X, if it exists, is $\begin{cases} \text{immediately followed in X by WRITE(addr,data)} \\ \quad \text{if there is an } i^{th} \text{ RTR(addr,--) in Y and it is RTR(addr,data)} \\ \text{last in X if there is no } i^{th} \text{ RTR(addr,--) in Y} \end{cases}$

This is easily seen to be ND-monotonic.

2.2 PACKET ACKNOWLEDGMENTS AND SAFETY

All of the systems considered so far have had to respond to incoming packets however fast they were sent by their user, and there was no limit to the rate at which the user could send them. In the first implementation of MEM, the memory unit has to accept the commands directly, and hence has to operate at unlimited speed. System #3, implementing NDMEM, seems a slight improvement in that it only has to put the commands into its buffers infinitely quickly, until one realizes that unless the memory unit itself is infinitely fast the buffers have to be infinitely large.

This is clearly unacceptable; no interconnection of speed-independent modules can make such assumptions. The problem is one of safety. No packet may be sent until its destination is ready to receive it. The safety problem arises at several levels in data flow computers. Here we are concerned with it only at its most microscopic level. The solution to the problem is to acknowledge each packet transmission. That is, for each port transmitting data, there is another port transmitting acknowledge packets in the opposite direction. Every data packet must be acknowledged before the next data packet can be sent on the same port. We will require all ports of all systems to have such an acknowledge port.

(Even systems which would be safe without acknowledge ports will have them. This is because of the manner in which packets are transmitted. A packet transmission is indicated by a zero to one transition of a "request" signal. An acknowledge signal from the receiver is needed to tell the transmitter to reset the request signal.)

The implementation of the system MEM may be modified to acknowledge input commands only after the transaction on the actual memory unit is completed. This will make it impossible for the user to send a command while the memory is busy. Of course, the output port must also have acknowledges, since the system to which the RTR packets are sent might be slow and need to be protected against overruns on its input. So the algorithm for AMEM (MEM with acknowledges) might be:

(1) If a WRITE packet is received, update the memory (take your time!)

and then send an acknowledge on the input acknowledge port.

(2) If a READ packet is received, fetch data from the memory and send a RTR packet out.

(3) If an acknowledge is received on the output acknowledge port, send an acknowledge on the input acknowledge port.

These three operations proceed concurrently and independently.

Transmission of acknowledge packets is behaviorally similar to transmission of normal packets, and can be handled in the same way in the specification of a system. That is, the acknowledge ports associated with output ports are treated exactly as though they were input ports, and vice-versa. The system AMEM has two input ports: the "real" input port X and the output acknowledge port Y_A; and two outputs: the "real" output port Y and the input acknowledge port X_A.

f_{AMEM}

input ports = X, Y_A output ports = Y, X_A

(1) $|Y|$ = number of READs in X

(2) Y_i = RTR(addr,data) where the i^{th} READ in X is READ(addr) and the last WRITE(addr,--) before it, if there is one, is WRITE(addr,data), or data = 0 if there is no WRITE(addr,--) before the i^{th} READ

(3) $|X_A| = |Y_A|$ + number of WRITEs in X

(4) $(X_A)_i$ = "ack"

(5) $|Y_A| \leq |Y| \leq |Y_A| + 1$

(6) $|X| - 1 \leq |X_A| \leq |X|$

It is easy to prove that the given implementation realizes parts (1), (2), (3), and (4) of f_{AMEM}. (It is very similar to MEM.) Parts (4), (5), and (6) constitute the "Standard Acknowledge Restriction" that we will require all systems and all users to obey.

Standard Acknowledge Restriction (S.A.R.) - weak form

If X is an input port and X_A is its acknowledge port,

(1) X_A consists only of "ack"

(2) $|X_A| \leq |X|$

If Y is an output port and Y_A is its acknowledge port,

(3) $|Y| \leq |Y_A| + 1$

Given that a system and its user both obey the weak form of the S.A.R., we can easily show that they obey the following:

Standard Acknowledge Restriction (S.A.R.) - strong form

If Z is an input or output port and Z_A is its acknowledge port,

(1) Z_A consists only of "ack"

(2) $|Z_A| \leq |Z| \leq |Z_A| + 1$

Proof: If Z is an input port of the system and an output port of the user, (1) and $|Z_A| \leq |Z|$ follow from the S.A.R. on the system (letting $Z = X$); and $|Z| \leq |Z_A| + 1$ follows from the S.A.R. on the user (letting $Z = Y$). If Z is an output port of the system and an input port of the user, just exchange "system" and "user".

The S.A.R. is clearly ND-monotonic and hence admissible as part of a functional specification.

In any proof that a system realizes a function, it suffices to show that it obeys the weak form of the S.A.R. contingent on its user obeying the strong form.

We can now prove that AMEM realizes parts (5) and (6) of f_{AMEM}, that is, the S.A.R. in strong form.

Let Y = output of AMEM and input to user, X = input to AMEM and output of user.

First, number of WRITEs in X

= number of acks sent on X_A in consequence of (1) of AMEM's implementation

= $|X_A|$ - number of acks sent on X_A in consequence of (3) of AMEM's implementation

= $|X_A| - |Y_A|$

Now $|Y|$ = number of READs in X (by (2) of AMEM's implementation)

= $|X|$ - number of WRITEs in X (by well-behavedness of user)

= $|X| - |X_A| + |Y_A|$ (derived above)

$\leq |X_A| + 1 - |X_A| + |Y_A|$ (from S.A.R. for user)

$\therefore |Y| \leq |Y_A| + 1$

Also $|X_A|$ = number of WRITEs in X + $|Y_A|$ (derived above)

$\leq$ number of WRITEs in X + $|Y|$ (from S.A.R. for user)

= number of WRITEs in X + number of READs in X (by (2) of AMEM's implementation)

= $|X|$

This proves the weak form of the S.A.R., from which the strong form follows.

2.2.1 CANONICAL PACKET COMMUNICATION

Since the Standard Acknowledge Restriction narrowly limits the way acknowledge ports are handled in the functional specification of a system, it is not uncommon for the handling of the acknowledge ports to be similarly limited in the implementation of the system. Wherever possible, system implementations will receive and transmit packets in the following way:

Canonical Packet Reception (RCVPKT)

(1) Wait until a packet has arrived on the input port (it might have already arrived by the time this step is executed); take its data

(2) Send an acknowledge for it

Canonical Packet Transmission (XMTPKT)

(1) Send the packet

(2) Wait for an acknowledge

These operations will appear in the system implementation language as "functions" that take port names as arguments and appear in assignment statements. The data conveyed by the := is the contents of the packet. Assignment statements containing these operations are like input/output operations in ordinary computer programs in that they "hang up" the program until the packet communication has taken place. "Var := RCVPKT(port)" waits until an incoming packet has arrived (and then acknowledges same). "XMTPKT(port) := expression" waits until the transmitted packet has been acknowledged. Programs may use multiprocessing as long as no RCVPKT or XMTPKT operations can be simultaneously executed by two processes on the same port.

It is easy to see that any implementation using the RCVPKT and XMTPKT operations obeys the Standard Acknowledge Restriction.

Systems need not use these canonical operations in order to be correct. For example, the implementation of AMEM given previously did not. That is why the proof that it obeyed the Standard Acknowledge Restriction w. so complicated.

Here is an implementation of CMEM, a system whose behavior is similar (but not identical) to AMEM:

```
process starts at A
input port X
output port Y
array memory init 0
var command, addr, data

A:      command := RCVPKT(X);
        if command = READ(--) then
            let command = READ(addr);
            data := memory(addr);
            XMTPKT(Y) := RTR(addr,data)
        else
            let command = WRITE(addr,data);
            memory(addr) := data;
        goto A
```

2.3 LATENCY

CMEM and AMEM behave differently in a subtle way. Suppose the user transmits a READ packet and then refuses to acknowledge the RTR packet that results. AMEM refuses to acknowledge the original READ, and the entire system comes to a halt, since the user can't send another command packet until the previous one was acknowledged. CMEM acknowledges the READ packet anyway (it happens automatically as part of the RCVPKT operation). It then refuses to acknowledge any further command packets until the RTR is acknowledged, because it gets hung up in the statement "XMTPKT(Y) := RTR(addr,data)". CMEM behaves as though it has an input buffer capable of storing one packet.

This difference shows up in the functional specification. Lines 2, 4, 5, and 6 of the specification of f_{AMEM} [section 2.2] apply to CMEM also. Lines 1 and 3 are different:

$$f_{AMEM}$$

(1) $|Y|$ = number of READs in X

(3) $|X_A| = |Y_A| + |X|$ - number of READs in X

$$f_{CMEM}$$

$$(1)\ |Y| = \begin{cases} \text{number of READs in X if } |X| = 0 \text{ or } 1 \\ \qquad\qquad \text{or } (\ |X| \geq 2 \text{ and } |Y_A| \geq \text{number of READs in (X - last packet))} \\ \text{number of READs in (X - last packet) otherwise} \end{cases}$$

$$(3)\ |X_A| = \begin{cases} |X| \text{ if } |X| = 0 \text{ or } 1 \\ \qquad\qquad \text{or } (\ |X| \geq 2 \text{ and } |Y_A| \geq \text{number of READs in (X - last packet))} \\ |X| - 1 \text{ otherwise} \end{cases}$$

This illustrates the fact that correct analysis of the latency of a system can be quite complicated and requires careful analysis of the algorithm.

The only difference between AMEM and CMEM arises if the user fails to acknowledge all RTR packets, that is, if $|Y_A| \neq |Y|$. If $|Y_A| = |Y|$, one can easily show that, for both AMEM and CMEM,

$|Y|$ = number of READs in X

$|X_A| = |X|$

(To prove this for CMEM, show that if $|X| \geq 2$, the case $|Y_A|$ < number of READs in (X - last packet) can't occur.)

The latency of a system is the number of commands that it can accept and acknowledge whose results have not been acknowledged by the user; that is, the number of pending commands that it can "remember". Because systems are so varied in their behavior, the concept of latency is not easy to define precisely.

One system for which it can be defined is the FIFO, or first-in-first-out buffer. A FIFO of length N (and having latency N) is a system with one input port and one output port, which realizes the identity function and acknowledges up to N more inputs than its user has acknowledged outputs. The function realized by a FIFO of length N is:

f_N

(1) $|Y| = \min \{ |X|, |Y_A| + 1 \}$

(2) $Y_i = X_i$

(3) $|X_A| = \min \{ |X|, |Y_A| + N \}$

A FIFO of length $N \geq 1$ can be implemented with a queue of size N and the following program:

<u>processes</u> <u>start</u> <u>at</u> A, B

<u>input</u> <u>port</u> X

<u>output</u> <u>port</u> Y

```
var m
var p init 0                          | queue population

A:      until p ≠ N do;
        k := RCVPKT(X);
        store k at end of queue;
        p := p + 1;
        goto A;

B:      until p ≠ 0 do;
        m := item taken from front of queue;
        XMTPKT(Y) := m;
        p := p - 1;
        goto B
```

For N = 1 this becomes:

```
process starts at A
input port X
output port Y
var P

A:      P := RCVPKT(X);
        XMTPKT(Y) := P;
        goto A
```

A FIFO of latency zero cannot be implemented by any system using the RCVPKT and XMTPKT operations, though it can be implemented with a few pieces of wire.

Appendix I contains a proof that a series connection of FIFO's of lengths M and N yields a FIFO of length M+N.

When systems differ only in their latency, it is sometimes possible to make them equivalent by adding FIFO's to various ports. For example, it can be shown that CMEM is identical to AMEM with a FIFO of length one on its input. If it could be shown that every system X is equivalent, except for latency, to a system X_0 defined as having latency zero, then the latency of the system X could be characterized by the lengths of the FIFO's that would have to be added to the various ports of X_0 to make it identical to X. A system of latency zero would have to be one which never acknowledges any input packet until all resulting output packets have been sent and acknowledged. AMEM is such a system, so CMEM could be said to have latency 1 on its its input port and zero on its output port. It is not clear whether such an analysis can be applied to nondeterminate systems of significant complexity.

2.3.1 ARBITRATORS, DISTRIBUTORS, AND ALLOCATORS

Three basic systems are very important in the design of the structure controller and memory, as well as other places in data flow computers.

The arbitrator is a nondeterminate system with N inputs and one output, which transmits each incoming packet to the output. The order of the packets from each input must be preserved in the output stream. The order in the output stream of packets from different ports is arbitrary. In any reasonable implementation it would depend on which input packet arrived first. An arbitrator realizes the following function, in which port number is indicated by a superscript instead of a subscript:

basic (zero latency) arbitrator f_{ARB}

If $X^1, X^2, \ldots X^N$ are inputs and Y is output,

$(X^1_A, X^2_A, \ldots X^N_A, Y) \in f_{ARB}(X^1, X^2 \ldots X^N, Y_A)$ if

(1) $|Y| = \min\left(\sum_{i=1}^{N} |X^i|, |Y_A| + 1\right)$

(2) $\forall\, i \in [1,N]$, $|X^i_A|$ = number of packets from X^i in first $|Y_A|$ packets of Y

(3) $\forall\, i \in [1,N]$, if $L(i) = |X^i|$, the sequence $\langle i, X^i_1\rangle, \langle i, X^i_2\rangle, \ldots \langle i, X^i_{L(i)}\rangle$ is a subsequence of Y.

Each incoming packet is tagged with its port number so that its source can be identified in the output. This identification feature is used in a few, but not all, applications of the arbitrator.

Arbitrators are the major component of the arbitration network of the data flow computer. The principal use of the arbitrator in the structure memory is to allow the address space to be divided into small pieces, with a separate memory module handling transactions on each piece. The LOAD packets sent back from the several modules are merged in an arbitrator, so that the entire interconnection of modules behaves as if it were one memory system.

Arbitrators of nonzero latency may be defined as zero latency arbitrators with various FIFO buffers on the ports. Such arbitrators are useful in various places throughout the data flow computer, but there is one place where the arbitrator must have latency zero. This is in the transmission of packets from the structure controller to the memory. When the structure controller receives an acknowledge for a packet it has sent to the memory, it must know that that packet is ahead of any other packets that might subsequently be sent to other input ports of the arbitrator on that memory unit. This problem will be explained in section 5.0.4.

An arbitrator of zero latency may be realized by the following program:

```
process starts at A
input ports X_1 . . . X_N
output port Y
var p, input

A:      wait until a packet is available on any input port,
                    let p := that port;
        | this is nondeterminate!
        input := the packet on port p;          | do not acknowledge yet
        XMTPKT(Y) := <p , input>;
```

send acknowledge on port p;
goto A

A distributor is a determinate system with one input and N outputs, which transmits incoming packets to the output port selected by a data field in the packet. Incoming packets are assumed to be of the form <port, data>. The distributor strips off the "port" field in the final result. An N-output distributor realizes the following function:

basic (zero latency) distributor f_{DIST}

If X is input and $Y^1, Y^2, \ldots Y^N$ are outputs,
$(Y^1, Y^2, \ldots Y^N, X_A) \in f_{DIST}(X, Y^1_A, Y^2_A, \ldots Y^N_A)$ if

(1) $\forall\, i \in [1,N],\quad |Y_i| =$ number of packets $<i,\text{--}>$ in X

(2) $|X_A| = \sum_{i=1}^{N} |Y^i_A|$

(3) $\forall\, i\ \forall\, j,\quad Y^i_j =$ data where j^{th} packet $<i,\text{--}>$ in X is $<i, \text{data}>$

Such a distributor may be implemented as follows:

process starts at A
input port X
output ports $Y_1 \ldots Y_N$

A: wait until a packet is available on port X;
z := the packet on port X; | do not acknowledge yet
let z = <port , data>;
XMTPKT(Y_{port}) := data;
send acknowledge on port X_A;
goto A

Higher latency distributors may be defined in terms of basic distributors and FIFO buffers.

Distributors are the principal component of the distribution network of the data flow computer.

An allocator is a nondeterminate variation of a distributor which transmits incoming packets to one of several output ports. Each packet is sent to any output port that is ready to receive it, that is, any port that has acknowledged all previous packets sent to it. An allocator is normally used to send packets to a group of identical units, always selecting any unit which is not busy. The structure controller of a data flow computer will typically be realized in the form of several identical units in order to increase throughput. Operation packets from the instruction cells will be sent through allocators to the structure control units. (In fact, the other functional units of a data flow computer will be handled the same way.) An N-output allocator realizes the following function:

basic (minimal latency) allocator f_{ALLOC}

If X is input and $Y^1, Y^2, \ldots Y^N$ are outputs,
$(Y^1, Y^2, \ldots Y^N, X_A) \in f_{ALLOC}(X, Y^1_A, Y^2_A, \ldots Y^N_A)$ if

(1) $\sum_{i=1}^{N} |Y^i| = |X|$

(2) $|X_A| = \min \{ |X|, N - 1 + \sum_{k=1}^{N} |Y^k_A| \}$

(3) $Y^1, Y^2, \ldots Y^N$ are disjoint subsequences of X

It may be implemented by the following program:

```
processes start at A, B
input port X
output ports Y^1 ... Y^N
queue q size N init (1, 2, ... N)
var pop init N

A.      wait until a packet is available on port X
```

z := the packet on port X; | do not acknowledge yet

k := item at head of q;

pop := pop - 1;

send packet z on port Y^k; | don't wait for acknowledge

until pop ≠ 0 do;

send acknowledge on port X_A;

goto A;

B: wait until acknowledge is available on any port Y_A^i,

let p := that port;

| nondeterminate!

take the acknowledge from port Y_A^p;

put p at end of q;

pop := pop + 1;

goto B

The basic allocator given above does not have latency zero in the sense of not acknowledging any input until the resultant output has been acknowledged - such an arrangement would defeat the allocator's purpose. The system given above does have the minimum latency that makes sense.

3.0 THE BASIC MEMORY MODULE

In this section a formal specification of the memory module "MM" will be given. MM is the fundamental building block of the packet memory system. Each MM system is a memory, somewhat like the system NDMEM described earlier, which handles a specific set of addresses. To increase total information transfer rate, the address space of the entire packet memory system may be divided into smaller pieces, with one MM unit handling each piece. The MM units are connected through arbitrators and distributors, and form a system which is itself an MM. This is "horizontal" composition, and is quite similar to the interleaving found in conventional memory systems. To increase the speed on individual transactions an MM unit may have a cache module "CM" connected to it. MM with CM connected to it is itself an MM. This is "vertical" composition, and is quite similar to the cache memories found in high performance conventional computers.

MM has one input port CMDI ("command in") taking command packets from its user, and one output port RESO ("result out") returning results to the user. The memory space is divided into <u>words</u> or <u>cells</u> (the terms will be used interchangably), each of which corresponds to one node of a structure. Every memory transaction refers to one word, and every incoming or outgoing packet bears the address of that word in its address field. The memory space is the same size as the address space, and the size is known to the user, so there can be no "nonexistent memory word" error. In most implementations, the memory size would be 2^N where the address field of every packet is N bits.

Notation: $FET^{(\pm)}$ means any of FET, FET^-, or FET^+. $LOAD^{(\pm)}$ similarly.

Each word in the memory contains a data field and a reference count field, which are used by the structure controller as described in section 1.2. $LOAD^{(\pm)}$ and UPD packets have corresponding fields. Furthermore, $FET^{(\pm)}$ packets have a tag field, which is returned unchanged in the corresponding $LOAD^{(\pm)}$ packet.

3.0.1 LATENCY AND INITIAL MEMORY CONTENTS

The specification of MM to be given below does not say anything about latency. This is because MM's user is required to acknowledge every result packet. When this happens, MM will acknowledge every command packet, regardless of what its actual latency is. Hence, an accurate description of MM's latency is unnecessary.

Initial memory contents will also be left unspecified. In the functional specification of a memory, the definition of initial contents arises in the specification of the system's response to a READ command that was not preceded by a WRITE. The specification of MM will assume that this does not occur. In an actual data flow computer, a free storage list will be generated when the system starts, which requires writing on every cell.

3.0.2 INFORMAL BEHAVIOR OF MM

There are 5 types of input packets to MM, and 4 types of output packets:

FET(addr, tag)	("fetch") reads the addressed word and returns LOAD(addr, data, ref, tag) ["ref" is the reference count]
FET^+(addr, tag)	increases the reference count by one and returns $LOAD^+$(addr, data, ref, tag) ["ref" is the reference count after the increment]
FET^-(addr, tag)	decreases the reference count by one and returns $LOAD^-$(addr, data, ref, tag)
CLR(addr)	("clear") waits until all FET/LOAD, $FET^+/LOAD^+$, and $FET^-/LOAD^-$ transactions on the indicated word have

completed, and then returns DONE(addr)

UPD(addr, data, ref) ("update") writes the data and reference count into the addressed word. It returns no result, and hence uses no tag.

MM is nondeterminate as was the example memory NDMEM, in that result packets referring to different cells are not constrained to appear in the same order as the commands that gave rise to them. MM is further nondeterminate in that it may rearrange $LOAD^{(\pm)}$ packets referring to the <u>same</u> cell. Such nondeterminacy would not have made sense for NDMEM, since RTR packets with the same data and same address were indistinguishable, but, in the case of MM, $LOAD^{(\pm)}$ packets may have different tags.

Since $LOAD^{(\pm)}$ packets involve a change of reference count and may be reordered arbitrarily, the question arises: What happens to the reference counts appearing in such packets if they are reordered? The answer is that the result packets have reference counts consistent with their own order, not the order of the original command packets.

Example: Suppose the reference count of cell A is 1, and the command sequence

$$FET^+(A, T1) ; FET^+(A, T2) ; FET^-(A, T3) ; FET^-(A, T4)$$

is sent. Some of the possible results are

$$LOAD^+(A, D, 2, T1) ; LOAD^+(A, D, 3, T2) ; LOAD^-(A, D, 2, T3) ; LOAD^-(A, D, 1, T4)$$

or

$$LOAD^-(A, D, 0, T3) ; LOAD^-(A, D, -1, T4) ; LOAD^+(A, D, 0, T1) ; LOAD^+(A, D, 1, T2)$$

The reference count temporarily becomes negative!

The reference count appearing in any $LOAD^+$ packet is one more than the count in the preceding $LOAD^{(\pm)}$ packet. Similarly, the count in a $LOAD^-$ is one less than, and the count in a LOAD is equal to, the count in the preceding LOAD$^{(\pm)}$. Some implementations of MM will never reorder LOAD$^{(\pm)}$ packets referring to the same address, although they may reorder those for different addresses. If this is the case, the reference count will never become negative, which removes the need for a sign bit in the reference count field.

3.0.3 INFORMAL BEHAVIOR OF MM'S USER

When the user gives a CLR command, it must not send any further commands of any type for the indicated cell, until the corresponding DONE packet has returned. (The purpose of the CLR command is to clear out pending transactions. It would defeat its purpose to continue sending commands.)

Like NDMEM, MM requires that no UPD command be given while any transactions are pending on the indicated cell.

3.0.4 FORMAL DEFINITION OF MM AND MMUSER

These definitions do not show latency or make any reference to acknowledges. The user is required to acknowledge every result packet and MM is consequently required to acknowledge every command. Both systems of course obey the Standard Acknowledge Restriction. The definitions do not consider the possibility of illegal packet types or invalid fields in packets. All universal quantifie. . are intended to range over a set that is in each case obvio from context.

Note: In rules 2, 3, and 4 the zeroth DONE in Y means the beginning of Y. The N+1st DONE in Y, where N = the number of DONEs in Y, means the end of Y. Similarly for CLRs in X. The intention is to let the DONE and CLR packets break up X and Y into intervals, which makes it convenient to think of the entire histories as being preceded and followed by DONE or CLR packets.

f_{MM}

If X is input and Y is output, $Y \in f_{MM}(X)$ if

(1) For all addr, the number of DONE(addr) packets in Y = the number of CLR(addr) packets in X

(2) For all addr, K, and tag, the number of LOAD(addr,--,--,tag) packets between the K^{th} and $K+1^{st}$ DONE(addr) in Y = the number of FET(addr,tag) packets between the K^{th} and $K+1^{st}$ CLR(addr) in X

(3) For all addr, K, and tag, the number of $LOAD^{-}$(addr,--,--,tag) packets between the K^{th} and $K+1^{st}$ DONE(addr) in Y = the number of FET^{-}(addr,tag) packets between the K^{th} and $K+1^{st}$ CLR(addr) in X

(4) For all addr, K, and tag, the number of $LOAD^{+}$(addr,--,--,tag) packets between the K^{th} and $K+1^{st}$ DONE(addr) in Y = the number of FET^{+}(addr,tag) packets between the K^{th} and $K+1^{st}$ CLR(addr) in X

(5) For all addr, J, and K, the J^{th} $LOAD^{(\pm)}$(addr,--,--,--) in Y is $LOAD^{(\pm)}$(addr,data,ref+D,--), where the last UPD(addr,--,--) before the J^{th} $FET^{(\pm)}$(addr,--) in X is UPD(addr,data,ref) and is preceded by I $FET^{(\pm)}$(addr,--) packets, and D = {number of $LOAD^{+}$(addr,--,--,--) packets} - {number of $LOAD^{-}$(addr,--,--,--) packets} among the $I+1^{st}$ to J^{th} $LOAD^{(\pm)}$(addr,--,--,--) packets in Y.

f_{MMUSER}

If Y is input to user and X is output, $X \in f_{MMUSER}(Y)$ if

(1) For all addr, either the number of CLR(addr) packets in X = the number of DONE(addr) packets in Y, or else there is one more CLR(addr) in X than DONE(addr) in Y, and there are no $FET^{(\pm)}$(addr,--) or UPD(addr,--,--) packets after the last CLR(addr) in X.

(2) For all addr, for any UPD(addr,--,--) in X, the number of $FET^{(\pm)}$(addr,--) packets preceding it is $\leq$ the number of $LOAD^{(\pm)}$(addr,--,--,--) packets in Y.

3.0.5 IMPLEMENTATION OF MM USING A RANDOM ACCESS DEVICE

Implementation of MM with a random access device is quite easy. Assume the memory is two arrays, mem-data and mem-ref, containing the data and reference count for each word, respectively. The following program will suffice:

```
process starts at A
input port CMDI
output port RESO
array mem-data, mem-ref
var command, addr, data, ref, tag

A:   command := RCVPKT(CMDI);

        if command = FET(--,--) then          | FET - return LOAD
           let command = FET(addr, tag);
           XMTPKT(RESO) := LOAD(addr, mem-data(addr), mem-ref(addr), tag)

        else if command = FET⁻(--,--) then    | FET⁻ - decrement ref and return LOAD⁻
           let command = FET⁻(addr, tag);
```

```
    mem-ref(addr) := mem-ref(addr) - 1;
    XMTPKT(RESO) := LOAD⁻(addr, mem-data(addr), mem-ref(addr), tag)

else if command = FET⁺(--,--) then    | FET⁺ - increment ref and return LOAD⁺
    let command = FET(±)(addr, tag);
    mem-ref(addr) := mem-ref(addr) + 1;
    XMTPKT(RESO) := LOAD⁺(addr, mem-data(addr), mem-ref(addr), tag)

else if command = UPD(--,--,--) then   | UPD - update memory
    command = UPD(addr, data, ref);
    mem-data(addr) := data;
    mem-ref(addr) := ref
else                                   | CLR - return DONE
    let command = CLR(addr);
    XMTPKT(RESO) := DONE(addr);

goto A
```

3.1 HORIZONTAL INTERCONNECTIONS OF "MM" SYSTEMS

The functional specifications of MM and its user have the useful properties that:

(1) f_{MM} and f_{MMUSER} are invariant under reordering of command packets referring to different words. That is, such a reordering will not affect the legal responses from MM, nor will it affect the legality of the commands from the user.

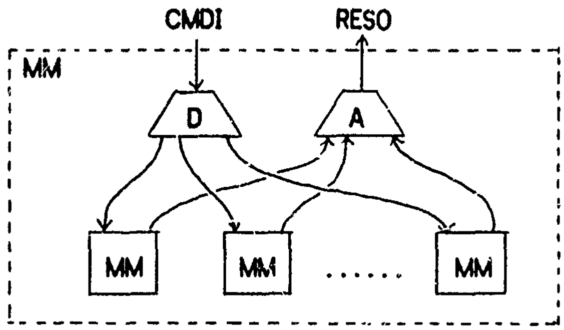
(2) f_{MM} and f_{MMUSER} are similarly invariant under reordering of result packets referring to different words.

(3) f_{MM} and f_{MMUSER} are invariant under reordering of $LOAD^{(\pm)}$ packets for the same word between any pair of DONE packets for that word, assuming the reference counts are suitably adjusted.

(4) the behavioral properties of MM and its user are completely independent for different words.

Property (4) makes it possible to connect MM systems and their users through distributors and arbitrators, and still have an MM system. The following connections are possible:

Multiple memory connection



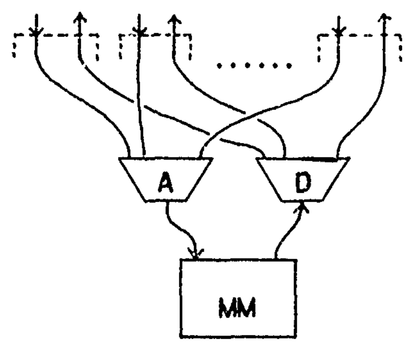
If each of the small boxes realizes f_{MM} (contingent on its user realizing f_{MMUSER}), the large dashed box realizes f_{MMUSER} for a larger address space. If the user of the large dashed box realizes f_{MMUSER}, each small box'es user realizes f_{MMUSER}.

For this to work the distributor and arbitrator must handle address fields properly. If there are 2^N small MM units, the address field of the interconnection is N bits longer than that of the units. The distributor picks out N bits of all incoming address fields and uses them as the output port numbers. (For interleaving purposes, it might be most effective to pick out the least significant bits.) Those bits do not appear in the address fields of the packets that are sent to the MM units. The arbitrator inserts the input port number of each incoming packet into the address field in the same positions as the bits that were removed by the distributor.

This connection is one of the methods by which the transaction rate can be increased. Random access memory devices have the property that every read or write transaction causes the device to become busy for some period of time, during which it cannot handle any other transactions. For example, a MOS RAM might be busy for 500 nanoseconds during every transaction, and therefore be able to handle 2 million transactions per second. Putting a FIFO buffer on it will increase its latency (as the term was defined previously), but its transaction rate stays the same. The only way to increase the data rate is to use many memory units. If a distributor can handle 64 million packets per second on its input port, and an arbitrator can handle 64 million packets per second on its output port, it might be reasonable to use 32 MOS RAM's, each in a separate MM unit. These are connected to a 32 port distributor and a 32 port arbitrator. The average rate at which packets come out of each port of the distributor is 2 million per second, which is the rate at which individual units can handle them. Assuming the commands are uniformly distributed over the address space, this interconnection will handle 64 million transactions per second. The retrieval delay for each item will still be 500 nanoseconds, but that is an unavoidable consequence of the memory technology used.

For this interconnection to work effectively, the latency of the individual MM units, or the output latency of the distributor, must be at least one, and preferably more. If the MM units and the distributor all have latency zero, the distributor will be unable to acknowledge a command, and hence unable to get the next one, until the command has been completely processed by the MM unit. This would defeat the purpose of using multiple units. In practice, the latency might be somewhat more than one, in order to maintain a transaction rate near the maximum in the presence of nonuniform statistical frequency of commands for each unit. This can be accomplished by placing a small FIFO buffer between the distributor and each MM unit.

Multiple user connection



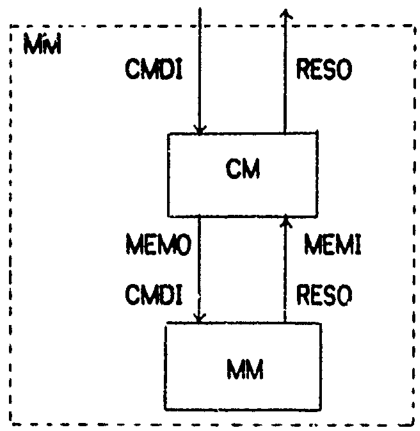
This is just like the multiple memory connection, but with the roles of MM and the user exchanged. If the solid box realizes f_{MM}, each of the interfaces at the top of the diagram realizes f_{MM} for a smaller address space. If each of the users of this interconnection realizes f_{MMUSER}, then the collection of all of them along with the arbitrator and distributor realizes f_{MMUSER} on the large address space.

As in the previous case, the arbitrator must map the input port number into a larger address field, and and distributor must remove the corresponding part of the address field and use it as the output port number. Each of the interfaces at the top of the diagram realizes an equivalent address space, and each uses a different subset of the memory space contained in the actual MM unit.

This connection would be used if there were several users, each presenting commands at such a slow rate that one memory module could handle all of them. Such a situation could arise if several cache modules are used which have a sufficiently high "hit" rate that the rate of memory requests arising from cache misses is low.

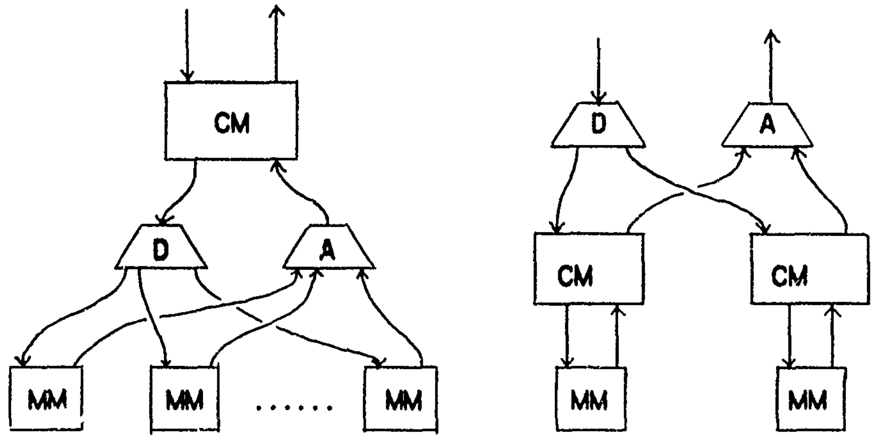
3.2 VERTICAL COMPOSITION AND THE CACHE MODULE

In the section we describe the cache module "CM" which connects to an MM system and, so connected, realizes an MM system with the same address space.



If the small box labelled MM realizes f_{MM}, the large dashed box realizes f_{MM}. If the user of the large dashed box realizes f_{MMUSER}, the user of the small box realizes f_{MMUSER}.

Vertical and horizontal interconnections may be mixed as in the following examples, since in each case the system being implemented is MM.



The purpose of a cache is to retain the data of a small subset of the main memory's address space, and return requests for data in that subset directly without reading it from main memory. Since the cache has much less data than the main memory, it can be built out of faster circuits and devices without being prohibitively expensive. Hence any request for a datum that is in the cache (a "cache hit") is answered very quickly. If the cache is sufficiently well designed that it has a high hit rate, the overall performance of the memory will be nearly as good as that of the cache itself.

A cache must be designed to maximize the hit rate by holding those memory items that are likely to be addressed. This is usually done by assuming that the addresses being used vary slowly with time, and so, when an item is referred to once, it is likely to be referred to again soon, and should be placed in the cache. Therefore, when an item is addressed which is not in the cache (a "cache miss"), the datum is fetched from main memory, placed in the cache, and also returned to the user. Subsequent requests for that datum will be cache hits.

The size of the "items" that the cache contains affect its performance. A cache for the main memory of a conventional computer may use rather large items consisting of, for example, 8 consecutive words. This is effective because references to memory, especially instruction fetches, tend to be localized in space. When a cache miss occurs on any word, a block of 8 consecutive words is read from main memory and loaded into the cache. Since references in the immediate future are likely to be in this block, the hit rate is increased.

The structure memory for a data flow computer has no such locality of reference. Therefore, the unit of cache organization will be the individual word.

Placing an item in the cache usually requires removing some other item. The most popular strategy, and the one that will be used here, is the "least recently used" (LRU) strategy. Each reference to a cache item is noted in some sort of reference table. When space must be made in the cache for a new datum, the item that has been used least recently, that is, has gone the longest time without a reference, is chosen.

When a write command is issued, the item in the cache is updated appropriately. In some cache organizations, the item in main memory is always updated also. This technique, known as "write through", will not be used here. Instead, the item in the cache will simply be marked as having been modified. When an item that has been modified must be displaced from the cache, it is first written into main memory. This method has a lower volume of commands going from the cache to main memory than the "write through" method.

It is crucial that the cache be able to determine very quickly whether or not it contains a given word. Since its memory space is much smaller than the full address space, it must store the full address with each item. When a command is received, the cache must be searched for an item with the given address. It is important that the search be conducted quickly.

A popular method of organizing the cache for rapid searching is the "set associative" memory [12]. The cache is organized as an array of columns and rows. The full address space is similarly organized, with the same number of columns, and a presumably much greater number of rows. Each item in the cache is constrained to correspond to the same column in the full address space as its own column in the cache. Therefore, to search for a given item whose full address is known, the address is separated into row and column. If it is in the cache, it must be in the same column as its column address in the real memory, so only that column of the cache need to be searched. Furthermore, only row addresses need to be stored in the cache along with the items. The column addresses are implicit from the position in the cache.

This organization works well for a suprisingly small number of rows in the cache. For example, the main memory cache on the IBM 370/168 computer has only four rows. (The number of rows is referred to as "cache depth".) To determine whether a given item is in the cache, only four address comparisons need to be made. These can easily be done simultaneously.

The column number of a word in the full address space is typically taken from the low bits of its address. The row number comes from the remaining bits. This allows consecutively addressed items to reside in the cache in adjacent columns of one row.

Example: Suppose the full address space contains 4096 addresses, and addresses consist of four octal digits. There are 8 columns, and the low digit of the address is the column number. The cache depth is three.

column number

	0	1	2	3	4	5	6	7
row address	551	550	543	504	444	425	425	425
data	A	B	C	D	E	F	G	H
row address	412	417	447	313	314	315	270	241
data	I	J	K	L	M	N	O	P
row address	242	242	242	242	246	271	365	413
data	Q	R	S	T	U	V	W	X

The cache holds the item with address 4472, with data "K". When a command is received requesting the contents of location 4472, the address is divided into the row (447) and the column (2). Column 2 of the cache is then searched for 447. It contains 543, 447, and 242. 447 is compared with these three numbers simultaneously. It matches the second of them, so the data associated with it (K) is returned to the user.

When a new item is to be put into the cache, its column number is known in advance, so only its row must be determined by searching the column for the least recently used item. For example, if an entry for 2124 must be created, column 4 is searched. If the least recently used item is 314, it is removed. If its "modify" bit is on, an UPD packet is sent to main memory, containing the address (3144) and the data (M). The row address is then changed to 212.

The determination of which item in a column was least recently used can be made by some simple scheme such as keeping a counter along with the data for each item. Whenever any reference is made, that item's counter is set to zero and all others in its column are increased by one. The least recently used item is the one with the highest count.

Because each operation in the cache involves examination of an entire column, the cache memory itself should be organized so that each column is a "word", that is, the entire column is read or written at once.

3.2.1 DESIGN OF CM

The functional specification of CM is very simple: it must realize f_{MM} through its "top" ports and realize f_{MMUSER} through its "bottom" ports.

f_{CM}

If (CMDI, MEMI) = input ports, and (RESO, MEMO) = output ports,
(RESO, MEMO) $\in f_{CM}$(CMDI, MEMI) if

(1) RESO $\in f_{MM}$(CMDI)
(2) MEMO $\in f_{MMUSER}$(MEMI)

An implementation of a system realizing f_{CM} will now be given. Each word of the full address space is in one of eight states denoted N, P, P', Q, Q', R, R', and T.

N - The word is not in the cache at all. (Since the cache is much smaller than the full address space, most words are in this state at any instant.) There are no pending commands from the user to the system. There are no pending commands from the cache to the main memory.

P - Space has been reserved in the cache for the word, and at least one FET$^{(\pm)}$ has been sent to main memory, but no LOAD$^{(\pm)}$ has come back. One or more FET$^{(\pm)}$/LOAD$^{(\pm)}$ transactions are pending to the cache. Exactly the same transactions are pending to the main memory.

P' - Same as P, but a CLR packet has been received from the user. One or more FET$^{(\pm)}$/LOAD$^{(\pm)}$ transactions, plus a CLR, are pending to the cache. The same transactions without the CLR are pending to the main memory.

Q - The first LOAD$^{(\pm)}$ has come back from main memory. A CLR packet will be sent as soon as main memory is able to accept it. Zero or more FET$^{(\pm)}$/LOAD$^{(\pm)}$ transactions are pending to the cache. Exactly the same transactions are pending to the main memory.

Q' - Same as Q, but a CLR packet has been received from the user. Zero or more $FET^{(\pm)}/LOAD^{(\pm)}$ transactions, plus a CLR, are pending to the cache. The same transactions without the CLR are pending to the main memory.

R - The word is in the cache, but some $FET^{(\pm)}/LOAD^{(\pm)}$ transactions may still be in progress in main memory. A CLR packet has been sent to remove them. No CLR packet has been received from the user. Zero or more $FET^{(\pm)}/LOAD^{(\pm)}$ transactions are pending to the cache. The same transactions, plus a CLR, are pending to the main memory.

R' - Same as R, but a CLR packet has been received from the user. Zero or more $FET^{(\pm)}/LOAD^{(\pm)}$ transactions, plus a CLR, are pending to the cache. Exactly the same transactions are pending to the main memory.

T - The word is truly in the cache. There are no pending transactions to the cache or from the cache to the main memory.

The normal states for a word are N or T, depending on whether the word is in the cache or not. In state T, all commands are acted upon immediately by the cache without any communication with main memory. In state N, any command from the user causes the word to undergo transitions that eventually result in its being in state T. If the command is a $FET^{(\pm)}$, the word must be read from main memory, and the state goes through some of the intermediate states. If the command is UPD, the word is created in the cache in state T. In either case, some other word may have to be displaced, going from state T to state N. If the "modify" flag for that word is on, an UPD packet is sent to main memory.

The specifications of MM and its user require that the user accept all result packets from MM. MM is only required to accept commands when the results of previous commands have been accepted by the user (although an efficient implementation of MM might allow many commands to be in progress at once.) Therefore, in order to avoid a deadlock, CM must accept packets from main memory, at MFMI, even when main memory refuses to accept any further commands through MEMO. CM sometimes must wait for memory to accept a command. While it is waiting, it may refuse to accept further commands at CMDI, but it must always be willing to accept packets at MEMI. CM may assume that any packet sent through RESO will be accepted.

The reason why CM allocates a cache cell for an item and puts it into state P as soon as the first $FET^{(\pm)}$ command comes from the user, is to avoid a deadlock, that is, a situation from which the system cannot proceed. If it simply sent the packet out through MEMO and did not allocate the cache cell until the first $LOAD^{(\pm)}$ packet came back, it would use its own space more efficiently, but would be in danger of deadlock. (P cells are useless, since they do not contain data.) This will be explained in section 6.0.

In the following description of the cache algorithm, the manipulation of the counters to determine the least recently used item is not shown.

STATE N

$FET^{(\pm)}$(addr, tag) at CMDI - Create space in the appropriate cache column. Either use an empty space (this situation can only arise when the system is first started) or remove the least recently used item in state T. If no item is in state T, wait until one enters state T, not accepting any packets on CMDI while waiting. (Items in other states will progress to state T.) When the item to be removed is found, write it out if its "modify" flag is on, by sending an UPD packet at MEMO. If main memory is not accepting packets at MEMO, wait until it does. Then create a new item in the cache with the given address, "modify" = 0, state = P. Leave the data and reference count fields unspecified. Also, send a $FET^{(\pm)}$ packet, identical to the incoming one, out through MEMO, to fetch the data.

CLR(addr) at CMDI - send DONE(addr) at RESO.

UPD(addr, data, ref) at CMDI - Create space in the cache as for $FET^{(\pm)}$, perhaps sending an UPD packet to memory. Then create a new item in the cache with the given address, "modify" = 1, data and reference count from the command, and state = T.

LOAD$^{(\pm)}$ or DONE at MEMI - can't occur because no transactions are pending in main memory.

STATE P

FET$^{(\pm)}$(addr, tag) at CMDI - Send the same packet at MEMO.

CLR(addr) at CMDI - Change to state P'.

UPD(addr, data, ref) at CMDI - can't happen, since transactions are pending in the cache.

LOAD$^{(\pm)}$(addr, data, ref, tag) at MEMI - Deposit the data and reference count into the cache word, and send the same packet out at RESO. If the main memory is accepting commands, send a CLR(addr) at MEMO and change this cache item to state R. If not, change to state Q.

DONE at MEMI - can't happen, since no CLR has been given to main memory.

STATE P'

FET$^{(\pm)}$, UPD, or CLR at CMDI - can't happen, since user has a CLR/DONE transaction pending.

LOAD$^{(\pm)}$(addr, data, ref, tag) at MEMI - Deposit the data and reference count into the cache word, and send the same packet out at RESO. If the main memory is accepting commands, send a CLR(addr) at MEMO and change this cache item to state R'. If not, change to state Q'.

DONE at MEMI - can't happen, since no CLR has been given to main memory.

STATE Q

Note: CM does not accept any command at CMDI whenever any item is in state Q. Q is simply a temporary state that is waiting to send a CLR(addr) out through MEMO and go into state R.

FET$^{(\pm)}$, UPD, or CLR at CMDI - can't happen, since cache is not accepting commands.

LOAD$^{(\pm)}$ at MEMI - same as state R.

DONE at MEMI - can't happen, since CLR has not been sent to main memory.

Main memory becomes able to accept a command - Send CLR(addr) through MEMO, change to state R.

STATE Q'

Note: CM does not accept any command at CMDI whenever any item is in state Q'. Q' is simply a temporary state that is waiting to send a CLR(addr) out through MEMO and go into state R'.

FET$^{(\pm)}$, UPD, or CLR at CMDI - can't happen, since cache is not accepting commands.

LOAD$^{(\pm)}$ at MEMI - same as state R.

DONE at MEMI - can't happen, since CLR has not been sent to main memory.

Main memory becomes able to accept a command - Send CLR(addr) through

MEMO, change to state R'.

STATE R

FET$^{(\pm)}$(addr, tag) at CMDI - Update the reference count in the cache, and set the "modify" bit if the packet was FET$^-$ or FET$^+$. Send LOAD$^{(\pm)}$(addr, data, newref, tag) through RESO, where data and newref are current contents of the cache. Note: at the instant this happens, there may still be FET$^{(\pm)}$/LOAD$^{(\pm)}$ transactions pending in main memory. If so, those FET$^{(\pm)}$ packets were earlier than this one, but the corresponding LOAD$^{(\pm)}$ packets won't be returned until later. This is the circumstance which causes the general system MM to occasionally return LOAD$^{(\pm)}$ packets in an order different from that of the FET$^{(\pm)}$ packets.

UPD(addr, data, ref) at CMDI - Update the cache, set the "modify" bit. Note: if an UPD packet is received while in state R, we know from the rules for MMUSER that no FET$^{(\pm)}$/LOAD$^{(\pm)}$ transactions are pending in main memory.

CLR(addr) at CMDI - Change to state R'.

LOAD$^{(\pm)}$(addr, data, ref, tag) at MEMI - Ignore the "ref" field in the packet. Increment or decrement the reference count in the cache if the packet is LOAD$^-$ or LOAD$^+$. Do not set the "modify" flag, since main memory already knows about the reference count change. Send LOAD$^{(\pm)}$(addr, data, newref, tag) through RESO, where newref = the updated reference count in the cache.

DONE(addr) at MEMI - Change to state T.

STATE R'

$FET^{(\pm)}$, UPD, or CLR at CMDI - can't happen, since user has a CLR/DONE transaction pending.

$LOAD^{(\pm)}$ at MEMI - same as state R.

DONE(addr) at MEMI - send DONE(addr) through RESO, change to state T.

STATE T

$FET^{(\pm)}$(addr, tag) at CMDI - Update the reference count in the cache, and set the "modify" bit if the packet was FET^{-} or FET^{+}. Send $LOAD^{(\pm)}$(addr, data, newref, tag) through RESO, where data and newref are current contents of cache.

UPD(addr, data, ref) at CMDI - Update the cache, set the "modify" bit.

CLR(addr) at CMDI - Send DONE(addr) through RESO.

$LOAD^{(\pm)}$ or DONE at MEMI - can't happen, since there are no pending transactions in main memory.

3.2.2 PROOF OF CORRECTNESS OF CM

A proof of CM's correctness is generally similar to that of the system MEM given in section 2.0.3. The memory state required in the specification is the contents of the last UPD packet in the input history. One must show that, for a cell in states Q, Q', R, R', or T, the data in the cache itself is the same as that in the last UPD packet at CMDI, and, if the modify bit is off, this data is in main memory also. For states N, P, and P', the correct data is in main memory, that is, the last UPD at CMDI has the same data as the last UPD at MEMO. These properties must be shown to be preserved for all state transitions, and it must be shown that all legal $FET^{(\pm)}$ commands will get the correct data. Furthermore, the effect of reference count modifications resulting from FET^{+} and FET^{-} commands must be taken into account.

4.0 IMPLEMENTATION OF MM USING A "ROTATING" DEVICE

"Rotating" memories such as charge coupled device (CCD) or "magnetic bubble" shift registers, or magnetic disks, are rightly considered to be essentially unusable for the main memory of a computer because of their excessive retrieval delay. In a data flow computer, total transaction rate is as important a criterion as retrieval delay, and so the disadvantages of these devices largely disappears, making them perhaps economical as a mass store. On the other hand, further improvements in RAM technology may render these shift registers obsolete for most applications. This section is predicated on the assumption that CCD's or bubble memories will be economical and useful in the packet memory system.

In a rotating memory, the data is structured in a ring which "rotates" past a "read/write head". Equivalently, one may think of it as a fixed ring and a pointer rotating around the ring, with memory transactions permitted only on the cell currently pointed to. If the addresses of words correspond to fixed places on the ring, it is possible to predict when any given cell will be pointed to. Commands from the user can be stored in a memory somewhat like a queue, sorted by position, so that the pending transaction at the head of the queue is always (or nearly always) the one that the pointer will reach next. This will make optimal use of the availability of data from the CCD.

There are a number of CCD architectures currently in use. In the "line addressed random access memory" (LARAM), only a small part of the device shifts at full speed at any one time. The rest shifts and recirculates at a much lower speed in order to conserve power. The intent is to make the device behave somewhat like a random access memory. To retrieve any one item, one finds the section in which that item is stored, and directs the CCD to shift that section at high speed until the desired item is found. While this is happening, the other sections are shifting much more slowly, so this architecture is not efficient when many items are being sought at one time. It is therefore not suitable for the type of packet memory system being considered here.

Two other types of CCD's are the "serpentine", which is simply a long shift register (it "snakes" back and forth on the IC chip), and the "serial-parallel-serial", which is simply a collection of interleaved shift registers. These two types differ only in engineering specifications such as data rate and power consumption. They both behave like long shift registers, and hence are suitable for the type of memory under discussion.

There are a number of implementation considerations that must be taken into account in designing a rotating packet memory. For example, a number of shift registers, one for each bit of a data word, may be used, so that a new data word comes into position on each clock pulse. On the other hand, a single shift register might be used, with each word stored serially, or any arrangement between these two extremes can be used. One might also use an unusual correspondence between address and shift register position. All of these considerations are irrelevant to the structure being considered, so we will assume the memory is a ring of full words, ordered by address, with address zero following the highest address, and the pointer scanning the ring in order of increasing address. Any other implementation is equivalent to this.

In the following, the memory will be referred to as the "CCD", regardless of what type of device it actually is.

Pending transactions (that is, packets received at CMDI) are stored in the <u>transaction list</u> (TL), which is presumably much smaller than the memory itself. The TL is presumably realized with a random access memory device. In order to avoid moving data in the TL unnecessarily, it has a ring structure just like the memory. Transactions are placed in the TL at or near the same angular position as the position in memory of the word to which they refer. Since the TL is a smaller ring than the memory, each address of TL corresponds to many consecutive addresses of memory.

Let ≪X≫ be the function mapping addresses in the entire address space into the corresponding address in the TL. This is called the <u>hash function</u> for reasons that will be explained later. ≪X≫ is just the integer part of the quotient of X divided by the ratio of memory size to TL size. In a realization in which all sizes are powers of two, ≪X≫ is just the appropriate number of high order bits of X.

When a command is received for address X, the command packet is placed in the TL at address ≪X≫, or the first free address thereafter if ≪X≫ is full. Assuming a uniform distribution of addresses appearing in commands, the TL should be uniformly filled. As the memory pointer rotates through the memory, another pointer, maintaining about the same angular position, rotates through the TL, picking out the next transaction to perform.

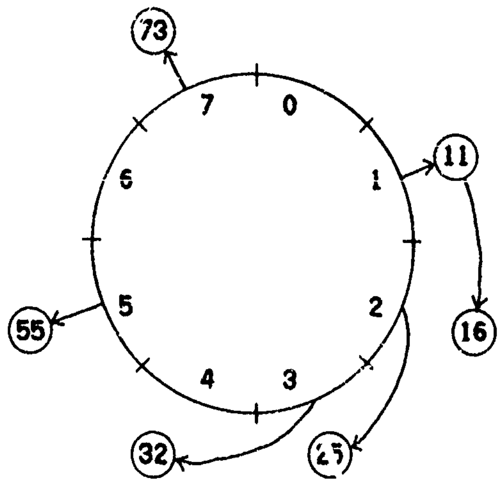
The TL is organized much like the "ordered hash table" devised by Amble and Knuth [2], with modifications to allow for its circularity and for the fact that items are being removed from it. In an ordered hash table, each item has a <u>hash address</u>. It is placed in the table at its hash address or in the contiguous block of items after the hash address. This block is in increasing order of data value. This ordering makes it possible to determine whether an item is in the table much more quickly than in a conventional hash table.

Although ordered hash tables are intended for entirely different applications than the transaction list of a packet memory, the concept is well suited to this application. The "value" of an item in the table is the word address appearing in the packet. Let a(P) denote this address for packet P, and call it the "CCD address". The "hash address" corresponding to CCD address X is just ≪X≫, defined earlier. (Hash functions are usually designed to be random, but that property is not desirable here.) The hash address of packet P is therefore ≪a(P)≫.

Because the TL is a ring instead of a linear list, a different definition of order is needed. The concepts of "greater than" and "less than" are replaced by "clockwise from" and "counterclockwise from". Since any item is both clockwise and counterclockwise from any other item, the order of two items must be defined relative to a third. This is done through the use of intervals denoted in ordinary mathematical notation. [X, Y] is the interval from X clockwise to Y. If $X \leq Y$, it has its customary meaning. If $X > Y$, [X, Y] is the set of numbers from X up to the highest address, and then from zero up to Y. "Open" and "half open" intervals have their customary meaning, that is, [X, Y) means [X, Y] exclusive of Y, etc. [X, Y) and [Y, X) are clearly complements of each other if $X \neq Y$.

The ordering of hash addresses and word addresses is expressed in terms of whether or not an element is in an interval. $Z \in [X, Y)$ means that if one starts at X and moves clockwise, one reaches Z before Y.

The general rule for maintaining order in TL is that, if one goes clockwise from an item's hash address to the item itself, one will not pass any empty cells and will pass only "smaller" items, that is, items whose hash addresses are counterclockwise from this one. This is best illustrated with a diagram. Let CCD addresses be two octal digits and hash addresses be one digit. The hash function picks out the first digit. The transaction list has 8 cells and is drawn as a circle.



Cells 0 and 6 are empty. Cell 2 contains a packet with address 16, whose hash address is 1 but was displaced because cell 1 is full.

It is possible for the transaction list to contain several packets referring to the same CCD address. Specifically, the following configurations are possible:

One or more $FET^{(\pm)}$ packets. When the CCD pointer reaches the appropriate address, its data will be read and sent back to the user in a sequence of $LOAD^{(\pm)}$ packets.

One or more $FET^{(\pm)}$ packets, followed by a CLR. When the CCD pointer reaches the appropriate address, the $LOAD^{(\pm)}$ packets will be sent out, followed by a DONE packet.

A single UPD packet. The data will be written into the CCD when the appropriate address is reached.

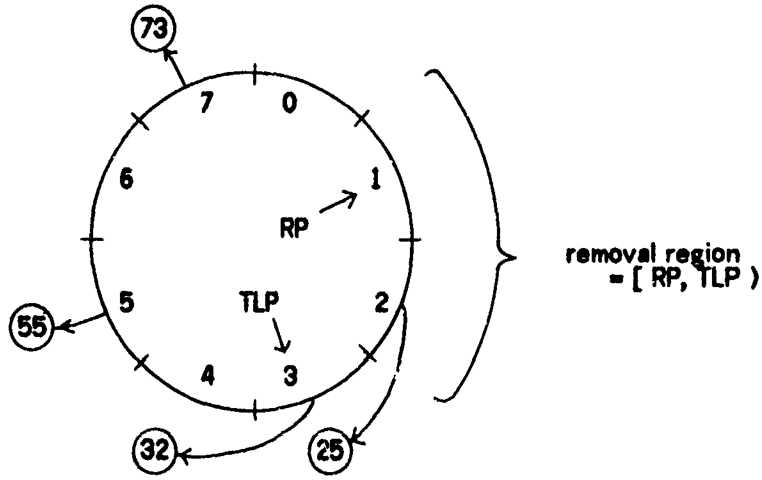
No other states are possible. This is because it is a violation of f_{MMUSER} to send an UPD packet when there are $FET^{(\pm)}$ or CLR packets pending. If an UPD is given when an UPD is already pending, the new one simply replaces the old one. If a $FET^{(\pm)}$ is given when an UPD is pending, the data is taken directly from the pending UPD packet and returned in a $LOAD^{(\pm)}$ packet.

Intuitively, the rule for a well formed transaction list is that the lines progressing clockwise from a cell to those items with that cell's hash address must never cross each other or pass over an empty cell. If an item with CCD address 43 were placed into cell 6, this rule would be violated, since the line from 4 to 43 would cross the line from 5 to 55. The insertion algorithm must instead put the 43 into cell 5 and move the 55 to cell 6. Furthermore, all items with the same hash address must be ordered by CCD address. In the example, 16 is clockwise from 11.

To insert an item, start at its hash address and search clockwise until an empty cell or a cell containing an item with higher (more clockwise) CCD address is found. In the former case, insert the new item. In the latter case, insert the new item after making space for it by pushing the old item, and all those contiguously following it, one space clockwise. In the example, insertion of item 10 would require pushing 11, 16, 25, 32, and 55 clockwise. Insertion of 42 would require pushing only the 55.

While incoming command packets are being placed in the TL by the above procedure, packets are being removed and sent to the CCD memory. This is accomplished through the use of a transaction list pointer (TLP) which rotates clockwise roughly in synchronization with the CCD address pointer. When the the CCD pointer points to CCD cell 10, the TLP points to TL address 1. Since a packet for address 11 is found there, it waits until the CCD pointer = 11, removes the packet from the TL, and performs the indicated operation on the contents of CCD address 11. The TLP is then immediately advanced to the next position, 2. Since the packet there specifies address 16, it waits until the CCD pointer = 16 and then removes the packet and performs the memory operation. The TLP the moves to 3 and the process continues.

The removal of items from TL makes it necessary to modify the rules for a well-formed transaction list. If 16 is removed from the example list, the line from cell 2 to item 25 passes through an empty cell, which would violate the condition given previously. Therefore, the region from which packets are removed is declared to be the "removal region", and it is permissible for the line from an item's hash address to the item itself to pass through the removal region. The removal region is delimited at its counterclockwise end by the "removal pointer" RP, and at is clockwise end by TLP. After removing 11 and 16, the example looks like this:



Whenever an item is removed, RP is set to the hash address of that item. In the example, after 25 is removed, RP will be set to 2 (25's hash address), and TLP will be advanced to 4.

The rules for a well-formed transaction list can now be given formally:

(1) $\forall$ j, k $\in$ TL address space, if $j \neq k$ and $TL(j) \neq \underline{\text{empty}} \neq TL(k)$,

$[\, \langle\langle a(TL(j)) \rangle\rangle , j \,] \;\not\subset\; [\, \langle\langle a(TL(k)) \rangle\rangle , k \,]$

(That is, the interval from the hash address of an item to the item itself is never contained within the corresponding interval for another item, i. e. the lines never cross.)

(2) $\forall\, j \in [\, RP , TLP \,)$, $TL(j) = \underline{\text{empty}}$

(That is, cells in the removal region are considered to be empty.)

(3) $\forall$ j, k $\in$ TL address space, if $TL(j) = \underline{\text{empty}} \neq TL(k)$ and $j \notin [\, RP , TLP \,)$,

$j \notin [\, \langle\langle a(TL(k)) \rangle\rangle , k \,]$

(That is, the interval from the hash address of an item to the item itself does not contain any empty cells not in the removal region.)

(4) $\forall$ j, k $\in$ TL address space, if $\langle\langle a(TL(j)) \rangle\rangle = \langle\langle a(TL(k)) \rangle\rangle$ and $j \in [\, \langle\langle a(TL(k)) \rangle\rangle , k \,]$,

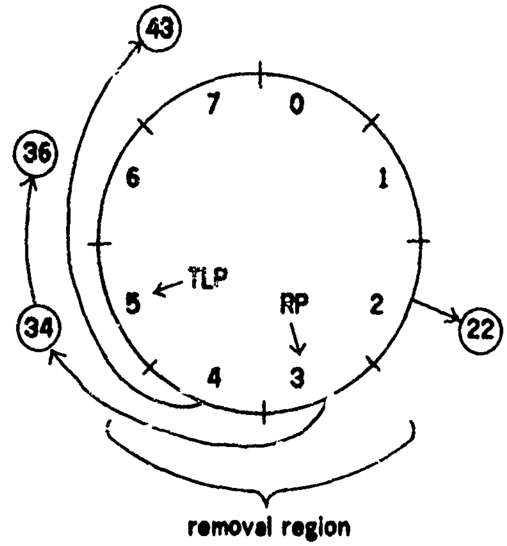
then $a(TL(k)) \geq a(TL(j))$

(That is, if two items have the same hash address, the more clockwise one has the higher CCD address, i.e. all the packets having one hash address are ordered by CCD address.)

(5) $\forall$ j, k $\in$ TL address space, if $j \in [\ \langle\!\langle a(TL(j)) \rangle\!\rangle, k\ )$ and $a(TL(j)) = a(TL(k))$, then $\forall\, m \in [\ j, k\ ]$, $a(TL(m)) = a(TL(j))$.
(That is, all items with one CCD address are adjacent. This is necessary to be sure that, when a sequence of adjacent FET$^{(\pm)}$ packets and a CLR are found, it is possible to return the LOAD$^{(\pm)}$ packets followed by a DONE, with no danger that there are unseen packets elsewhere referring to the same CCD address.)

(6) $\forall$ j, k $\in$ TL address space, if $j \in [\ \langle\!\langle a(TL(j)) \rangle\!\rangle, k\ )$ and $a(TL(j)) = a(TL(k))$, then TL(j) was placed in the table before TL(k)
(That is, the items with the same CCD address are ordered by age, the youngest being most clockwise.) This property makes it possible to return a DONE packet as soon as a CLR is encountered in the removal scan, since the packets are encountered in the same order as they were originally received.

The insertion algorithm requires some care when passing through the removal region. If the scan starts outside of the region and then enters the region, the item is placed in the first cell, and the region is shortened by one so that that cell is no longer part of the region. If the scan begins in the region but not in its first cell, the scan skips over the region and starts after its end. If the scan begins in the first cell of the region, it skips to the end if its CCD address is greater than or equal to that of the item just past the end. Otherwise, it is inserted in the first cell and the region is shortened.



To insert:	Do this:
22-27	put at 3, set RP := 4
30-33	put at 3, set RP := 4
34-35	put at 6, push the 36 and 43
36-42	put at 7, push the 43
43-77, 00-07	put at 0

The algorithm for inserting an item into the TL is given in appendix III A. If the TL already contains an UPD packet for the same address, it instead performs the indicated action, perhaps modifying the UPD packet and perhaps transmitting a packet at RESO.

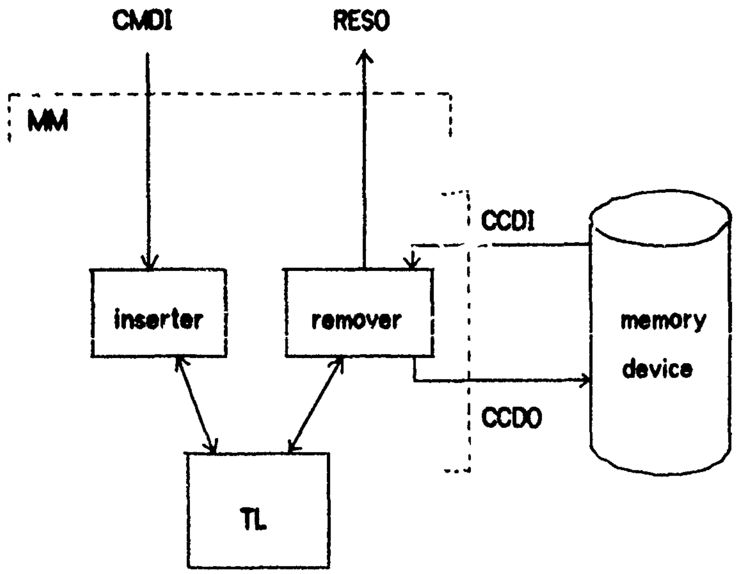
The removal algorithm is somewhat simpler. The TL item pointed to by TLP is next to be removed. The CCD pointer indicates the current item available at the CCD output. From the standpoint of the algorithms for handling the TL, the CCD pointer must be considered to be inexorably advancing under control of an external agency. The external agency is the clock controlling the shifting of the CCD shift register, or, in the case of a magnetic disk memory, it is the information being read from the disk's timing tracks.

The fact that the CCD pointer is synchronized to external events means that it cannot be integrated fully into a system using the packet communication principle. It must be considered external to the packet system, and some synchronizers or arbitration devices must be used in the interface. The design of such an interface is a common problem of digital system design, and is beyond the scope of this thesis. We will assume that the interface between the synchronous memory device and the packet system consists of ports CCDI and CCDO. Every time the CCD advances to a new address, an ADDR packet containing that cell's address and data are sent to the system through port CCDI. If the system fails to acknowledge the ADDR packets fast enough, so that the CCD is prevented from sending one, it may either drop the packet or wait until the CCD has shifted all the way around to the same address again. After the system receives an ADDR packet at CCDI announcing that an address has been reached, it may transmit a WRITE packet at CCDO, giving the address and new data to write. If this packet is not transmitted soon enough, it might be too late to write the data into the CCD. In this case, the CCD shifts all the way around, not emitting any ADDR packets, until the address is reached again, and then writes the data.

Wasting an entire rotation time whenever the asynchronous part of the system can't keep up with the CCD clock may seem drastic, but it doesn't happen very often. Whenever an asynchronous system must communicate with something such as the CCD clock, there is the possibility that it may be late. However, it is not difficult to design the system such that the probability of this happening is vanishingly small. If this is done, it is possible to prescribe drastic remedies when it does occur, without significantly degrading system performance.

The above description of the interface to the CCD may be somewhat simple-minded. Many memory devices require that the write command, and the data to be written, be given before the previous data from the same address is available. This means that the protocol whereby the system issues a WRITE packet only after receiving an ADDR packet bearing the data might not be appropriate. In the case of a CCD or other shift register, the problem can be solved by having two "taps" on the register: one for reading, and another, one or two bits later, for writing. In the case of a disk memory, the problem is more serious, and may require that the disk announce each address slightly before the data becomes available. The necessary modifications to the asynchronous part of the system will not be treated here.

The rotating memory module then looks like this:



The removal algorithm waits for an ADDR packet at CCDI matching the address contained in the packet in the transaction list pointed to by TLP. When found, it performs the indicated transaction, perhaps sending a packet out at RESO. It then sets RP to the hash address of the item which was just processed, which may shorten the removal region. The item is then erased from the transaction list, and TLP is advanced to the next position. If TLP now points to an item having the same CCD address, that item is processed also, using the same data. All transactions giving the same address are handled in this way. Any reference count changes are noted, and the modified reference count is written back into memory with a WRITE packet at CCDO.

When TLP reaches a cell which does not contain a transaction for the same address, either it is for a different address or it is empty. In the former case, the system waits for the CCD to reach the new address. In the latter case, it sets RP = TLP, destroying the removal region, and then advances both RP and TLP, in step with the ADDR packets that give the CCD address, until it finds a transaction to perform.

The algorithm for the rotating memory is given in appendix III B.

5.0 STRUCTURE CONTROLLER DESIGN CONSIDERATIONS

In this section we will examine a few of the considerations that must go into the design of an efficient structure controller.

5.0.1 CHECKING THAT THE CONTROLLER OBEYS F_{MMUSER}

The structure controller never issues an UPD command unless the reference count is known to be one. Since this is so, there can be no transactions pending on that cell, so the requirements of f_{MMUSER} are met. This is contingent, of course, on the rest of the computer correctly realizing $f_{CONTROLLERUSER}$. A reference count violation by the computer could lead to an UPD packet being sent while there are transactions pending.

5.0.2 PRECISE REFERENCE ACCOUNTING WITH IMPRECISE REFERENCE COUNTS

In checking that f_{MM} satisfies the needs of the structure controller, there is a point of possible danger that needs to be checked. Since $LOAD^{(\pm)}$ packets may be returned from the memory in an order different from that of the $FET^{(\pm)}$ packets, it was shown in section 3.0.2 that the reference counts returned from the memory may be unusual, perhaps even negative. Is it possible for this to interfere with the cell management mechanism? The answer is no, as long as the following rule is obeyed:

> After increasing a reference count (with a FET^+), do not pass the result to any destination until the corresponding $LOAD^+$ has returned.

For example, if an instruction cell indicates two destinations for its result, the reference count of the result must be increased with a FET^+ before the result is sent to the destination cells. If one of those cells is a SELECT that issues a FET^- to reduce the reference count, the FET^+ must act first. Furthermore, it is not enough to rely on the zero latency arbitrator to be sure the FET^+ gets to the memory before the FET^-. The FET^- must not be sent until the $LOAD^+$ arising from the FET^+ has returned. This is accomplished by not sending the result to the destination cells until the $LOAD^+$ has been received.

It is easy to see that no cell will fail to be reclaimed that should be reclaimed. At the time the last "owner" of a cell issues a FET^- to discard it, there are no other operations pending on the cell, so the $LOAD^-$ packet that is returned will have the correct reference count, which is zero.

To see that no cell will be accidentally reclaimed that shouldn't be, consider a cell with reference count 2, owned by instruction cells X and Y. Suppose X performs a structure operation that discards its copy, so that a FET^- is issued. We must show that if Y does not discard its copy, the $LOAD^-$ that arises from X's operation will not have reference count zero. The only way the reference count could possibly go to zero is if Y also causes a FET^-. Since Y does not intend to discard its copy of the cell, a FET^+ must have been issued first. (That is, the reference count should actually go up to 3, then down to 2 and then 1.)

The memory receives the following sequence at CMDI:

$$FET^-(addr, X) \;\; ; \;\; FET^+(addr, Y) \;\; ; \;\; FET^-(addr, Y)$$

The situation to be avoided is that in which the second and third LOAD packets are reversed:

$$LOAD^-(addr,--, 1, X) \;\; ; \;\; LOAD^-(addr,--, 0, Y) \;\; ; \;\; LOAD^+(addr,--, 1, Y)$$

This can't happen, because the $FET^-(addr, Y)$ is not sent until the $LOAD^+(addr,--,--,Y)$ has been returned.

5.0.3 MEMORY LATENCY

MM's latency was left unspecified only for the purpose of proving correctness of MM and its user. When actually implementing a practical packet memory, it may be necessary to build a high degree of latency into some modules in order to obtain good performance. For example, a "rotating" implementation of MM using a charge coupled shift register may be designed to have hundreds or thousands of commands pending at one time, although its correctness does not depend on this.

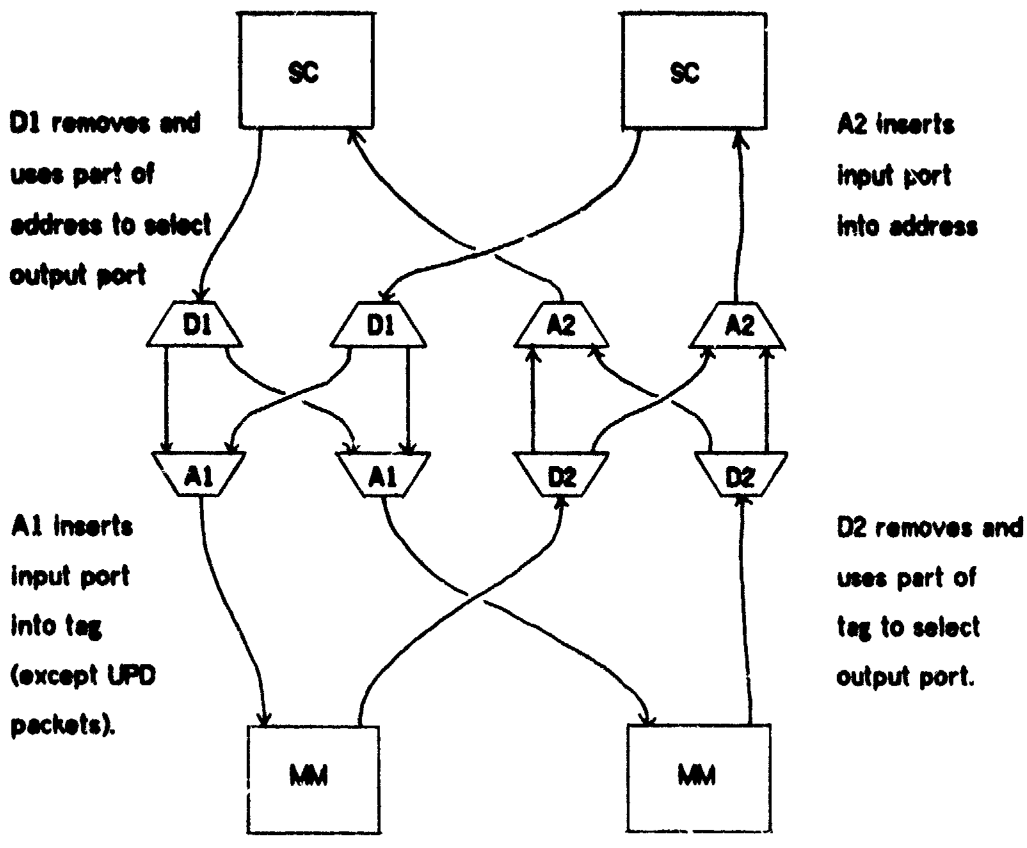
5.0.4 THROUGHPUT AND DISTRIBUTED PROCESSING

One of the fundamental principles of data flow computers is that, if enough parallelism exists in the program, a computer be able to run arbitrarily fast for a given logic speed. To do this, it must distribute the computation and be free of bottlenecks. If a data flow computer could only have one multiply unit, that would be a bottleneck, since it would limit the rate at which multiplies could be performed. The data flow concept must not place any restrictions at all on the number of multipliers that a computer can have (although any given computer of course has a fixed number). There must not even be bottlenecks in ports through which packets must pass. If every multiply operation packet had to pass through one input port of an allocator on its way to the multipliers, that would be unacceptable, since the logic speed places a limit on the rate at which packets can pass through a port. For example, if a port could handle packets 100 times faster than a multiplier could process them and all packets had to pass through one port, it would mean that no more than 100 multipliers could be usefully employed.

In the case of simple functional units such as multipliers, it is not difficult to avoid bottlenecks. Multiple functional units may be used, and the arbitration and distribution networks that connect them to the instruction cells may be designed to be free of bottlenecks and thus maintain any desired throughput rate [5]. For the same reason, multiple structure controllers are used, each with its own ports connected to the arbitration and distribution networks of the data flow computer. Also, multiple memory units are used, because the total memory transaction rate is greater than can pass through a single pair of CMDI/RESO ports.

It is not possible to compartmentalize the structure operation facilities as can be done with simple functional units. Connecting each structure controller to one memory module is not correct, because each structure controller must have access to the entire memory address space. The structure controllers must be connected to the memories through an <u>interconnection network</u> consisting of arbitrators and distributors for packets going in each direction. Command packets from the structure controllers have part of the address field removed and used to select the output port of the distributor, just as was done for the multiple memory connection in section 3.1. In this way, each structure controller "sees" the full address space, while each memory module supports only a small part of the total address space. The command packets from the different structure controllers are merged in arbitrators, which append the incoming port number to the tag field, so that the result packet will be returned to the correct controller. Packets coming out of the RESO ports of the memory modules pass through distributors that use the added bits of the tag field, and arbitrators that use the incoming port number to reconstruct the full address.

Interconnection network



The treatment of address fields and tag fields is symmetrical. One could think of all pending structure operations as occupying a "tag space". Just as each memory module supports a small part of the total address space, each structure controller supports a small part of the total tag space. The job of the interconnection network is to make the entire address space available to each structure controller, and to make the entire tag space available to each memory unit.

It is not necessary for the network to place the distributors before the arbitrators. Such a network would have a size proportional to the product of the number of structure controllers and the number of memory units, which may be excessive. It is possible to mix arbitrators and distributors in a network in such a way that the size is reasonable but bottlenecks are avoided.

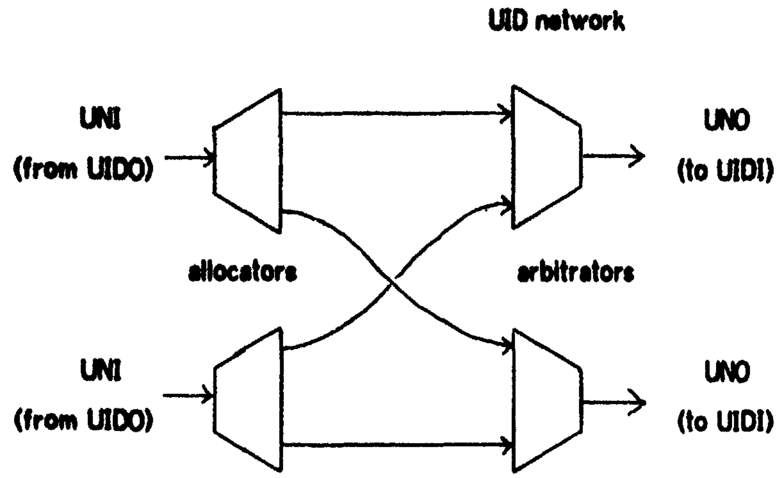
Because UPD packets do not have a tag field and do not give rise to result packets at RESO, it is necessary that the arbitrators and distributors carrying packets from the structure controllers to the memory modules (those labelled A1 and D1 in the preceding diagram) have latency zero. This is so that, when a structure controller receives an acknowledge for an UPD packet, it will be guaranteed that the packet has passed through the arbitrator and is therefore ahead of any packet that may subsequently be introduced into another input of the arbitrator. Suppose this were not done: One structure controller might write on a cell, thereby completing the creation of a structure. When it receives an acknowledge for that UPD command, it assumes that the structure is complete, and so it returns it to the rest of the computer. An instruction cell in the computer, having received this structure, may fire, causing a SELECT operation to be generated. The allocator may send the SELECT operation packet to another structure controller, which then sends out a FET packet with the same address. If there is buffering before the arbitrator that merges packets from the two structure controllers, the original UPD packet might still be in such a buffer, so the FET packet passes through the arbitrator first. If this happens, the old data will be read, rather than the new data supplied by the UPD packet. By making sure that the distributor and arbitrator have latency zero, the UPD packet cannot get stuck in a buffer. When the first structure controller receives an acknowledge for the UPD packet, that packet is known to have been accepted by the arbitrator, and hence it will precede any subsequent FET packet.

It it is not feasible for the interconnection network to use distributors and arbitrators that have no memory, it is necessary to put tag fields in all UPD specification passing through the network. An "adapter unit" is placed between the network and each memory module. The adapter passes all packets through except UPD packets. When it receives UPD(addr, data, ref, tag), it sends UPD(addr, data, ref) to the memory and UACK(tag) back to the interconnection network. The structure controller does not return a structure to the rest of the computer until it has received UACK replies for all UPD commands that it has sent. Whether such UACK packets are required is a question of the design of efficient routing networks and is beyond the scope of this thesis.

5.0.5 THE FREE STORAGE LISTS

To maintain just one free storage list would create a bottleneck, so each structure controller has one. Whenever a structure controller needs a word in order to create a node, it takes its address from the packet presented at input port UIDI. (UID stands for unique identifier.) The structure controller does not ask for addresses at UIDI; they are supplied in an "unending" stream, as fast as they are acknowledged.

The sources of the streams at UIDI are also the structure controllers, each of which maintains a free storage list and sends out addresses through output port UIDO. The UIDO ports are connected to the UIDI ports through a collection of allocators and arbitrators called the UID network. The purpose of this network is to maintain a supply of free cells to all controllers, even if some controllers' free storage lists should run out.



Each structure controller, in addition to performing structure operations, maintains a free storage list. Whenever an acknowledge is received on UIDO, it takes a cell from the list and transmits it in a UID packet through UIDO. Since a reference count scheme is used for recovering unused cells, the controller watches for words whose reference counts go to zero. Every time it reduces a reference count by issuing a FET$^-$ command, it examines the LOAD$^-$ packet that is returned. If it shows a reference count of zero, the word is reclaimed. This involves placing the word in the free storage list and, since whatever pointers it contained are destroyed, reducing their reference counts if their elem bits are off. If either or both of the latter reference counts go to zero, those words are reclaimed by the same process.

The procedure is recursive, and is an unpleasant type of recursion because the completion of each operation can produce two more operations to perform. Although the recursion always terminates, a huge amount of storage may be required to hold the list of words that need to have their reference counts reduced. The problem at its worst can be observed in the case of a large tree, no subtree of which is shared with anything else, whose root node is discarded. All nodes have an initial reference count of 1, so, when each node has its count reduced, it goes to zero, making it necessary to reduce the counts of both of that node's offspring.

To implement this procedure by simply issuing two FET$^-$ packets whenever a word's reference count goes to zero (that is, whenever a $LOAD^-$ is received bearing a count of zero), would create an intractable deadlock problem because of the proliferation of packets. Instead, the procedure that should be used is that only the right offspring of a word should be treated at the time the word is placed on the free storage list. The pointer to the left offspring will remain in the word while it is on the free storage list. The recursion in this procedure is under control, since only one new operation is created for every operation that is completed. When a word is taken from the free storage list, the reference count of its left offspring is reduced, which may cause one or more words to be reclaimed, before the word is used.

The memory management algorithm is as follows:

(1) Whenever a word's reference count is reduced, examine the $LOAD^-$ packet that is returned. If it shows a count of zero, put the word on the free storage list and, if the elem bit in its right half is zero, reduce the reference count of the word pointed to by that half. This may cause this step to be repeated.

(2) Whenever an acknowledge is received from port UIDO, get a word from the free storage list and send the packet UID(addr, its left half) through UIDO. (The contents of the left half are sent simply to avoid an extra memory reference.)

(3) Whenever a fresh cell is needed for creation of a structure node, take the packet UID(addr, obj) at port UIDI and acknowledge them. Addr is the address of the new cell. If the elem bit of obj is off, reduce the reference count of the addressed word. This may cause step (1) to be invoked.

5.0.6 MAINTAIN'NG INTEGRITY OF THE REFERENCE ACCOUNTING MECHANISM

The possibility of an error in the reference accounting and cell management mechanism is a troublesome problem, because, as explained in section 2.1.1, it is impossible for the memory to detect a reference accounting error by its user. Furthermore, the effects of such an error are unpredictable, and may show up in completely unrelated parts of the computation. However, there are a few things that can be done to minimize the probability of such an error being undetected.

First, all cells on the free storage list can be marked in some way, perhaps by a bit reserved for this purpose. Any reference to a marked cell other than for the purpose of removing it from the free storage list is a detectable error. Also, the free storage list can be organized in such a way that cells are added at one end and removed from the other, thereby maximizing the time that a cell stays on the list once it is put there. If a cell is erroneously reclaimed while a "spurious" pointer to it exists, it will then probably still be on the free storage list when the spurious pointer is used, so the error can be detected.

Another way of checking integrity of reference counts is to conduct an "audit" of the entire computer. This can be done at the end of the computation, and at any point during the computation. The host computer must disable all instruction cells and wait for all pending operations to clear out of the structure controllers and the routing networks. All reference counts can then be checked against the contents of the input registers of the instruction cells.

6.0 THE DEADLOCK PROBLEM

The structure controller and cache module that were described previously were both required to have a large capacity for state information which would be unnecessary if one could always be sure that the device lower in the hierarchy would accept a command.

In the case of the structure controller, the general behavior upon receiving a result packet from the memory is to perform some transformation on the data in its state memory and then send a new command packet. Its internal state memory could be dispensed with, and the state information placed directly into the tag fields of the packets. When a result packet is received from the memory, a "memoryless" controller's functions would then be simply to perform a transformation on the packet itself, forming a new packet which is sent to the memory. The reason this fails is that one can't be sure the memory won't decide to return several result packets (perhaps all pending ones) before it accepts any more command packets. Suppose this happened to a memoryless structure controller. It would have no place to put the result packets if the memory unit isn't accepting any more commands, so a deadlock would occur. The problem is that the controller has violated the rule that it must always be prepared to accept the results of all pending operations. A structure controller having state memory avoids this problem by always having space to absorb the results of all pending operations.

A similar problem arises in the cache module. If a word is not in the cache and a $FET^{(\pm)}$ packet is received, a cell is immediately allocated for it and placed in state P. A $FET^{(\pm)}$ packet is also sent to main memory to fetch the data. Until the data returns from the memory, the cell in the cache does not have data in it, so it serves no useful purpose. It might seem to make more sense to allocate the cache cell only when the first $LOAD^{(\pm)}$ packet is received from the memory rather than when the first $FET^{(\pm)}$ packet is received from the user - that is, to bypass state P altogether. The problem is that the creation of a cell in the cache may require writing out the cell's former contents. If the cell is created in consequence of the $LOAD^{(\pm)}$ packet coming from memory, the cache may have to send a packet to memory in response to a packet from memory. If the memory sends such $LOAD^{(\pm)}$ packets but does not accept any replies, the cache would have no place to put the data, so a deadlock would occur. The cache implementation given in section 3.2 avoids this problem by reserving space for the $LOAD^{(\pm)}$ packet in advance. If an UPD packet must be sent to the memory, it is done in response to input from the user rather than from the memory. This way, if the memory temporarily refuses to accept the UPD, the cache can simply refuse to accept input from its user.

In both the structure controller and the cache, the cost incurred as a result of this problem is an amount of memory equal to all the packets that can be simultaneously pending in all lower levels. In the controller, this is the state information for all concurrently executing structure operations. In the cache, a cell might be in state P for every $FET^{(\pm)}/LOAD^{(\pm)}$ cycle that is pending at that instant, Since a cell in state P is useless, the cache must be that much larger than it otherwise would be, for a given level of performance.

In the case of the structure controller, the memory space is needed somewhere in any case. If a great number of memory transactions can be pending simultaneously, a "rotating" memory, such as was described in section 4.0, is presumably being used. If a memoryless structure controller is used, the state information for pending operations is stored in the tag fields instead of the controller. But the tags of pending memory operations must be stored in the transaction list of the rotating memory, so whatever space was saved in the controller is used up in the transaction list.

Why, then, would a memoryless structure controller be more desirable? The reason is that memory space inside the controller is much more expensive than in the transaction list. The controller must be able to process information as fast as the highest level of the memory hierarchy. If that highest level is a cache using high speed (and expensive) devices, the controller must be equally fast. The rotating memory is at the bottom of the hierarchy, so its transaction list can use a slower and less expensive logic family.

In order to use a memoryless structure controller or a cache which does not use "P" cells, the memory system below the controller or the cache must obey the following "fixed latency law":

Whenever a result packet is transmitted at RESO, the device must accept a packet at CMDI. If that packet is an UPD, it must accept yet another, until it has taken one that is not UPD. It must do this even if the user does not accept anything further at RESO.

The reason UPD packets are a special case is that they do not generate any result, so the system should be able to absorb them in unlimited numbers.

Some memory systems obey this law. A random access implementation of MM clearly does. A rotating implementation can also, since the transaction list has fixed size. Whenever an item is taken out of the TL, another can be inserted. (The implementation of the rotating memory given in section 4.0 did not always behave this way, but it could easily be modified to do so.)

The systems that do not obey the fixed latency law are the horizontal composition of MM units and the cache. The former includes the interconnection network between the structure controllers and the memory units. In the case of the horizontal interconnection of units each of which obeys the fixed latency law, when one unit transmits a result packet, it will accept a new command. That result packet passes through the arbitrator and becomes a result of the interconnection, so the interconnection must accept another command. If the command is addressed to a different MM unit than the one that transmitted the result, that unit might not be able to accept it. What is needed is a way for the units to share the burden of pending transactions with each other.

In the case of the cache, maintaining a constant number of pending transactions in the cache and memory combined requires maintaining a constant number of pending transactions in the memory alone. For every result packet transmitted by main memory, another command must go from the cache to main memory. However, such commands only occur when there are cache misses. If the cache runs into unusually good luck and gets a continuous string of cache hits, it would not send commands to memory. In order to maintain constant latency, it would have to refuse any result packets from memory. This could result in some transactions remaining pending indefinitely. While this probably won't cause a data flow computer to malfunction, it might be an undesirable effect in general.

These difficulties can probably be overcome through the addition of extra circuitry to be sure that there is always space to handle all packets. It is not clear whether the benefits of a memoryless structure controller and a cache without state "P" justify such measures.

7.0 SUGGESTIONS FOR FURTHER RESEARCH

One of the principal problems remaining in the area of the design of systems using the packet communication principle is the development of a practical and systematic procedure for constructing modules that can be proven to meet given functional specifications. An important tool for this task is the development of a rigorous and concise Architecture Description Language (ADL). With the help of the ADL, the task can be divided into two parts:

(1) Development of a proof methodology so that systems expressed in the ADL can be proven to meet functional specifications.

(2) Development of a system construction methodology so that systems expressed in the ADL can be constructed with confidence that the physical device will realize the ADL expression.

For this purpose, the ADL must be simple enough to correspond neatly to the hardware devices involved, but powerful enough to make proofs involving history arrays tractable.

Another remaining problem is, of course, to develop functional specifications for all parts of the data flow computer system, including the structure controller, and give proofs of their correctness. The functional specification of the computer itself (that is, the structure controller's user) is needed, among other things, to show that no reference count violations will occur.

An efficient structure controller needs to be designed, with special attention to the needs of programs that are likely to arise.

The deadlock problem needs to be examined carefully, to see if it is worthwhile to build a memoryless structure controller.

REFERENCES

1. Ackerman, W. B. Interconnections of Determinate Systems. Computation Structures Group Note 31, Laboratory for Computer Science, MIT, July 1977.

2. Amble, O., D. E. Knuth. Ordered Hash Tables. The Computer Journal 17, (May 1974), pp 135-142.

3. Anderson, D. W., F. J. Sparacio, R. M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction Handling. IBM J. Res. and Dev. 11, 1 (Jan. 1967), pp 8-24.

4. Berkeley, E. C., D. G. Bobrow. The Programming Language LISP, its Operation and Applications. MIT Press, 1966.

5. Boughton, G. A. Routing Networks in Packet Communication Systems. S. M. Thesis in Preparation. Department of Electrical Engineering and Computer Science, MIT.

6. Dennis, J. B., D. P. Misunas. A Preliminary Architecture for a Basic Data Flow Processor. Computation Structures Group Memo 102, Laboratory for Computer Science, MIT, Aug. 1974.

7. Dennis, J. B., D. P. Misunas, C. K. Leung. A Highly Parallel Processor Based on the Data Flow Concept. Computation Structures Group Memo 134, Laboratory for Computer Science, MIT, Jan. 1977.

8. Dennis, J. B. Packet Communication Architecture. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, Aug. 1975.

9. Keller, R. M. Look-Ahead Processors. ACM Computing Surveys 7, 4, (Dec. 1975), pp 177-195.

10. Leung, C. K. Architecture Description Language. Computation Structures Group Memo in preparation, Laboratory for Computer Science, MIT, Aug. 1977.

11. Leung, C. K. Formal Properties of Well-Formed Data Flow Schemas. MAC TM66, Department of Electrical Engineering and Computer Science, MIT, June 1975.

12. Madnick, S. E., J. J. Donovan. Operating Systems. McGraw Hill, 1974.

13. McCarthy, J. et. al. LISP 1.5 Programmer's Manual. MIT Press, 1966.

14. Patil, S. S. Closure Properties of Interconnections of Determinate Systems. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York, 1970, pp. 107-116.

15. Rumbaugh, J. E. A Parallel Asynchronous Computer Architecture for Data Flow Programs. MAC TR150, Department of Electrical Engineering and Computer Science, MIT, May 1975.

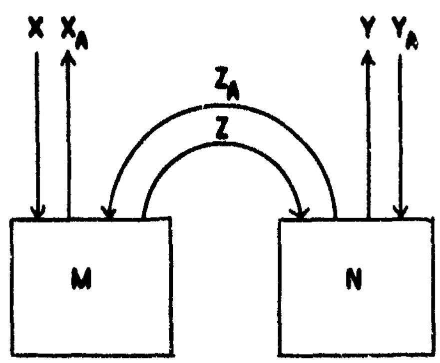
16. Thurber, K. J., L. D. Wald. Associative and Parallel Processors. ACM Computing Surveys 7, 4, (Dec. 1975), pp 215-255.

APPENDIX I

Proof that the concatenation of two FIFO buffers is a FIFO buffer, and lengths are additive.

This proof is given not because the statement is of fundamental interest, but as an example of the method of proving theorems about the behavior of systems, showing acknowledgments in detail.

Let a FIFO of size M have input port X and output port Z.
Let another FIFO of size N have input port Z and output port Y,
and let the ports Z and the acknowledge ports Z_A be linked.



From the definition of the first FIFO,

$$(1)\ |Z| = \min\{\,|X|,\ |Z_A| + 1\,\}$$
$$(2)\ Z_i = X_i$$
$$(3)\ |X_A| = \min\{\,|X|,\ |Z_A| + M\,\}$$

From the definition of the second FIFO,

$$(4)\ |Y| = \min\{\,|Z|,\ |Y_A| + 1\,\}$$
$$(5)\ Y_i = Z_i$$
$$(6)\ |Z_A| = \min\{\,|Z|,\ |Y_A| + N\,\}$$

Case I: Suppose $|X| \leq |Y_A| + N$

By the strong form of the Standard Acknowledge Restriction,

$$\text{either} \quad |Z| = |Z_A| \quad \text{or} \quad |Z| = |Z_A| + 1$$

If $|Z| = |Z_A| + 1$, then

$|Z_A| = |Y_A| + N$ (from 6, since $|Z_A| \neq |Z|$)

$|Z| \leq |X|$ (from 1)

$\therefore\ |Z_A| < |X|$

$\therefore\ |Y_A| + N < |X|$, which is a contradiction, so we must have $|Z| = |Z_A|$

$|Z| = |X|$ (from 1, since $|Z| \neq |Z_A| + 1$)

$\therefore\ |Y| = \min \{ |X|, |Y_A| + 1 \}$ (from 4)

$|X_A| = \min \{ |X|, |X| + M \}$ (from 3)

$\therefore\ |X_A| = |X|$ (since $M \geq 0$)

$|X| \leq |Y_A| + M + N$ (by hypothesis and fact that $M \geq 0$)

$\therefore\ |X_A| = \min \{ |X|, |Y_A| + M + N \}$

Case II: Suppose $|X| > |Y_A| + N$

If $|Z| = |Z_A|$, then

$|Z| = |X|$ (from 1, since $|Z| \neq |Z_A| + 1$)

$|Z| \leq |Y_A| + N$ (from 6)

$\therefore\ |X| \leq |Y_A| + N$, which is a contradiction, so we must have $|Z| = |Z_A| + 1$

$|Z_A| = |Y_A| + N$ (from 6, since $|Z_A| \neq |Z|$)

$\therefore\ |Z| = |Y_A| + N + 1$

$\therefore\ |Y_A| + 1 \leq |Z|$ (since $N \geq 0$)

$\therefore\ |Y| = |Y_A| + 1$ (from 4)

$|Z| \leq |X|$ (from 1)

$\therefore\ |Y_A| + 1 \leq |X|$

$\therefore\ |Y| = \min \{ |X|, |Y_A| + 1 \}$

$|X_A| = \min \{ |X|, |Y_A| + M + N \}$ (from 3 and $|Z_A| = |Y_A| + N$)

In either case,

$$|Y| = \min \{ |X|, |Y_A| + 1 \}$$

$$Y_i = X_i \qquad \text{(from 2 and 5)}$$

$$|X_A| = \min \{ |X| , |Y_A| + M + N \}$$

which are the conditions for the interconnection being a FIFO of length M + N.

APPENDIX II

Algorithm for the cache.

Actual lookup in the cache is not shown. Instead, the special functions cache-data(addr), cache-ref(addr), cache-state(addr), and cache-mod(addr) are used. These are treated as though they were arrays, and are assumed to be defined whenever the given address exists in the cache. In-cache(addr) returns true if the given address exists in the cache.

Can-create(addr), where addr does not exist in the cache, tells whether it can be created, that is, whether some cell in its column is unused or is in state T.

If can-create(addr) is true, creation-cell-is-empty(addr) tells whether the former case holds, and, if so, cache-create(addr) performs the insertion into an unused cell. Otherwise, cell-to-displace(addr) returns the address of a cell in state T, selecting the least recently used item. Cache-rename(old, new) performs the replacement.

```
processes start at Q, A
input ports CMDI, MEMI
output ports RESO, MEMO
var cmd, item, addr, data, ref, old-addr, p
var m init false          | tells whether to wait for input from MEMI
var memoflag init true    | true when last packet sent at MEMO has been acknowledged
var memowait init false   | true when need to send something on MEMO
var wait-pkt              | the thing to send
var create-flag init false    | true when need to create a new cache cell
var create-pkt            | command that led to creation
var new-addr              | address field of create-pkt
```

```
Q:
wait for acknowledge on port MEMO_A;
take the acknowledge;
memoflag := true;
goto Q


A:
until memoflag or packet is available on port MEMI do;
m := false;             | becomes true if should take packet at MEMI
if memoflag then        | is memory ready for command?

   if some-cell-is-in-state-Q-or-Q' then      | see if need to send a CLR
      addr := address-of-a-cell-in-state-Q-or-Q';
      memoflag := false;
      send CLR(addr) on port MEMO;
      if cache-state(addr) = "Q" then      | change Q to R, Q' to R'
         cache-state(addr) := "R"
      else
         cache-state(addr) := "R' "

   else if memowait then       | see if need to send FET(1) after creating a cell
      memowait := false;
      memoflag := false;
      send wait-pkt on port MEMO

   else if create-flag then       | see if trying to create a cell
      if can-create(new-addr) then    | is some cell in its column empty or in state T?
         create-flag := false        | yes, will create the cell
         if creation-cell-is-empty(new-addr) then
            cache-create(new-addr)      | old cell empty, just put in new address
         else
```

```
            old-addr := cell-to-displace(new-addr)     | find cell to displace
            if cache-mod(old-addr) then
                memoflag := false     | write out previous contents if necessary
                send UPD(old-addr, cache-data(old-addr), cache-ref(old-addr)) on port MEMO;
            cache-rename(old-addr, new-addr);     | create the new cell

| the new cache cell now exists

        if create-pkt = UPD(--,--,--) then     | what command caused the creation?
            let create-pkt = UPD(--, data, ref);     | UPD, fill in new cell appropriately
            cache-mod(new-addr) := true;
            cache-data(new-addr) := data;
            cache-ref(new-addr) := ref;
            cache-state(new-addr) := "T"
        else                    | command was FET(±)
            cache-mod(new-addr) := false;
            cache-state(new-addr) := "P";
            wait-pkt := create-pkt;     | queue command for transmission to memory
            memowait := true
    else
        m := true       | can't create new cache cell, must wait
else
    wait for packet on MEMI or CMDI, let P = that port;
    if p = 'CMDI' then

| +++++ process packet from CMDI +++++

        cmd := RCVPKT(CMDI);
        if cmd = FET(±)(--,--) then
            let cmd = FET(±)(addr, tag);
            if in-cache(addr) then
                if cache-state(addr) = "P" then
```

```
            memoflag := false;    | state P, just send it onward
            send cmd on port MEMO
        else              | state is R or T
            if cmd = FET+(--,--) then   | need to update reference count?
                cache-ref(addr) := cache-ref(addr) + 1;
                cache-mod(addr) := true;
                XMTPKT(RESO) := LOAD+(addr, cache-data(addr), cache-ref(addr), tag)
            else if cmd = FET-(--,--) then
                cache-ref(addr) := cache-ref(addr) - 1;
                cache-mod(addr) := true;
                XMTPKT(RESO) := LOAD-(addr, cache-data(addr), cache-ref(addr), tag)
            else
                XMTPKT(RESO) := LOAD(addr, cache-data(addr), cache-ref(addr), tag)
    else          | state N
        new-addr := addr;       | set flags so cell will be created
        create-pkt := cmd;
        create-flag := true
else if cmd = UPD(--,--,--) then
    let cmd = UPD(addr, data, ref);
    if in-cache(addr) then      | must be state R or T
        cache-data(addr) := data;
        cache-ref(addr) := ref;
        cache-mod(addr) := true
    else        | state N
        new-addr := addr;       | set flags so cell will be created
        create-pkt := cmd;
        create-flag := true
else              | must be CLR
    let cmd = CLR(addr);
    if in-cache(addr)       | state P, R or T
        if cache-state(addr) = "R" then
            cache-state(addr) := "R' "
```

```
            else if cache-state(addr) = "P" then
                cache-state(addr) := "P" *
            else            | state T
                XMTPKT(RESO) := DONE(addr)
        else            | state N
            XMTPKT(RESO) := DONE(addr)

| ++++++ end of CMDI processing ++++++

    else
        m := true           | packet was from MEMI

else
    m := true;          | memoflag was off, must handle MEMI input

if m then

| +++++ process packet from MEMI +++++

    item := RCVPKT(MEMI);
    if item = LOAD(±)(--,--,--,--) then
        let item = LOAD(±)(addr, data, ref, tag);
        if cache-state(addr) = "P" then    | know it is in cache
            cache-data(addr) := data;
            cache-ref(addr) := ref;
            XMTPKT(RESO) := item;
            if memoflag then        | can send packet at MEMO?
                memoflag := false;      | yes
                send CLR(addr) on port MEMO;
                cache-state(addr) := "R"
            else
                cache-state(addr) := "Q"       | no
```

```
    else if cache-state(addr) = "P' " then
        cache-data(addr) := data;
        cache-ref(addr) := ref;
        XMTPKT(RESO) := item;
        if memoflag then        | can send packet at MEMO?
            memoflag := false;      | yes
            send CLR(addr) on port MEMO;
            cache-state(addr) := "R' "
        else
            cache-state(addr) := "Q' "      | no

    else            | must be state Q, Q', R, or R'
        if item = LOAD+(--,--,--,--) then   | update ref and send LOAD
            cache-ref(addr) := cache-ref(addr) + 1;
            cache-mod(addr) := true;
            XMTPKT(RESO) := LOAD+(addr, data, cache-ref(addr), tag)
        else if item = LOAD-(--,--,--,--) then
            cache-ref(addr) := cache-ref(addr) - 1;
            cache-mod(addr) := true;
            XMTPKT(RESO) := LOAD-(addr, data, cache-ref(addr), tag)
        else
            XMTPKT(RESO) := LOAD(addr, data, cache-ref(addr), tag)
else                    | must be DONE
    let item = DONE(addr);
    if cache-state(addr) = "R" then     | know it is in cache
        cache-state(addr) := "T"
    else            | must be state "R' "
        cache-state(addr) := "T";
        XMTPKT(RESO) := DONE(addr);

| ++++++ end of MEMI processing ++++++
```

goto A

APPENDIX III A

The insertion algorithm for the rotating memory.

```
flag = false      | becomes true if TL already has UPD packet for this address

P := «a(X)»      | scan pointer = hash address initially
if RP ≠ TLP and P = RP and        | hash addr = start of removal region?
              («a(X)» ≠ «a(TL(TLP))» or a(X) < a(TL(TLP))) then
   TL(P) := X;                    | insert item at P
   RP := RP + 1 mod M;            | shorten the removal region
   pop := pop + 1                 | update TL population
else
   if RP ≠ TLP and P ∈ [RP, TLP) then    | hash address in removal region
      P := TLP                    | advance to end of removal region

| repeat until find empty cell or enter removal region

   until (P = RP and RP ≠ TLP)   or   TLP(P) = empty   or   flag = true   do
      (

| see if TL already has UPD with same CCD address

      if a(X) = a(TL(P)) and TL(P) = UPD(--,--,--) then
         flag := 1;
      else
         if («a(TL(P))» = «a(X)» and a(X) < a(TL(P)))
                    or «a(X)» ∉ [ «a(TL(P))» , P ]   | is X "smaller" than the current item?
         then
            Y := TL(P);        | save item from TL
            TL(P) := X;        | insert X here
            X := Y;            | insert saved item in next cell
                                    (which pushes everything past here)
```

```
        P := P + 1 mod M      | advance P to next cell
     );

| find out whether to insert X or process it directly

   if not flag then                | insert it
      if P = RP and RP ≠ TLP          | entered removal region?
      then
         TL(P) := X;               | insert item at P
         RP := RP + 1 mod M        | shorten the removal region
      else
         TL(P) := X;               | insert item at P
      pop := pop+1                 | update TL population

   else                            | process it directly
      let TL(P) = UPD(addr, data, ref);
      if X = UPD(--,--,--) then
         TL(P) := X             | another UPD, new one replaces old
      else if X = FET(--,--) then
         let X = FET(--,tag);   | FET, get the data
         XMTPKT(RESO) := LOAD(addr, data, ref, tag)
      else if X = FET+(--,--) then
         let X = FET+(--,tag);   | FET+, get the data and update ref
         TL(P) := UPD(addr, data, ref+1);
         XMTPKT(RESO) := LOAD+(addr, data, ref+1, tag)
      else if X = FET-(--,--) then
         let X = FET-(--,tag);   | FET-, get the data and update ref
         TL(P) := UPD(addr, data, ref-1);
         XMTPKT(RESO) := LOAD-(addr, data, ref-1, tag)
      else                           | must be CLR
         XMTPKT(RESO) := DONE(addr)
```

APPENDIX III B

The rotating memory algorithm.

```
process starts at A
input ports CMDI, CCDI
output ports RESO, CCDO
var P, X, Z, addr, data, ref, tag, CCD-addr, pop init 0, TL-cmd,
                    CCD-data, CCD-ref, CCD-newref, TLP, RP
array TL size M

A:    if TL(TLP) = empty then
         RP := TLP;          | destroy the removal region
      while TL(TLP) = empty and TLP ≠ «CCD-addr» do
      (
         TLP := TLP + 1 mod M;       | advance until catch up to CCD-addr
         RP := TLP                   | keep removal region destroyed
      );

      | look for input packets

      if pop ≥ M - 1
      then           | TL nearly full, can't take packets at CMDI
         Z := RCVPKT(CCDI);     | wait for and accept packet at CCDI
         let Z = ADDR(CCD-addr, CCD-data, CCD-ref);
         CCD-newref := CCD-ref
      else           | can accept packet on either port
         wait for packet at CMDI or CCDI, set P := that port      | nondeterminate!
         if P = 'CCDI' then
            Z := RCVPKT(CCDI);     | accept packet at CCDI
            let Z = ADDR(CCD-addr, CCD-data, CCD-ref);
            CCD-newref := CCD-ref
         else
```

```
        X := RCVPKT(CMDI);        | take packet at CMDI
        | ++++++++++++++++++++++++++++
        | + insert or otherwise dispose of X
        | +   (from appendix III A )
        | ++++++++++++++++++++++++++++

| perform all transactions matching CCD-addr

while TL(TLP) ≠ empty and a(TL(TLP)) = CCD-addr do
(
    TL-cmd := TL(TLP);        | remove transaction from list
    TL(TLP) := empty;
    pop := pop-1;             | update TL population
    RP := ≪a(TL-cmd)≫;            | shorten removal region appropriately
    TLP := TLP + 1 mod M;
    if TL-cmd = CLR(CCD-addr) then
        XMTPKT(RESO) := DONE(CCD-addr)

    else if TL-cmd = FET(--,--) then
        let TL-cmd = FET(addr, tag);
        XMTPKT(RESO) := LOAD(addr, CCD-data, CCD-newref, tag)

    else if TL-cmd = FET+(--,--) then
        let TL-cmd = FET+(addr, tag);
        CCD-newref := CCD-newref + 1;
        XMTPKT(RESO) := LOAD+(addr, CCD-data, CCD-newref, tag)

    else if TL-cmd = FET-(--,--) then
        let TL-cmd = FET-(addr, tag);
        CCD-newref := CCD-newref - 1;
        XMTPKT(RESO) := LOAD-(addr, CCD-data, CCD-newref, tag)
```

```
    else            | must be UPD
        let TL-cmd = UPD(addr, data, ref);
        XMTPKT(CCDO) := WRITE(addr, data, ref)
};

| rewrite reference count if it has changed

if CCD-ref ≠ CCD-newref then
    XMTPKT(CCDO) := WRITE(CCD-addr, CCD-data, CCD-newref);

goto A
```

Official Distribution List

Defense Documentation Center
Cameron Station
Alexandria, Va 22314 12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, Va 22217 2 copies

Office of Naval Research
Code 102IP
Arlington, Va 22217 6 copies

Office of Naval Research
Code 200
Arlington, Va 22217 1 copy

Office of Naval Research
Code 455
Arlington, Va 22217 1 copy

Office of Naval Research
Code 458
Arlington, Va 22217 1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, Ma 02210 1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, Il 60605 1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, Ca 91106 1 copy

New York Area Office
715 Broadway - 5th floor
New York, N. Y. 10003 1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375 6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380 1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, Ca 92152 1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Mathematics Department
Bethesda, Md 20084 1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350 1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-91T)
Office of Chief of Naval Operations
Washington, D. C. 20350 1 copy